Dash+: Extending Alloy with Hierarchical States and Replicated Processes for Modelling Transition Systems

Tamjid Hossain and Nancy A. Day
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada N2L 3G1
Email: {t7hossain, nday} @uwaterloo.ca

***Abstract*—Modelling systems abstractly shows great promise to uncover bugs early in system development. The formal language Alloy provides the means of writing constraints abstractly, but lacks explicit constructs for describing transition systems. Extensions to Alloy, such as Electrum, DynAlloy, and Dash, provide such constructs. However, still missing are language constructs to describe easily multiple processes with the same behaviour (replicated processes) running in parallel as is found in languages such as PlusCal and PROMELA. In this paper, we describe our proposal for adding explicit constructs to Dash for replicated processes. The result is Dash+: an Alloy language extension for describing transition systems that include both concurrent and hierarchical states and parametrized concurrent processes.**

***Index Terms*—Alloy, Statecharts, declarative modelling, transition systems, replicated processes**

I. Introduction

Modelling systems abstractly and declaratively has become an important tool for conquering complexity and discovering bugs early in system development. Difficult problems, such as cache coherence and consensus protocols, can be modelled abstractly and investigated for problems prior to more detailed UML modelling and coding. Early abstract modelling languages such as Z [1] and VDM [2] evolved to languages such as Alloy [3] and TLA+ [4], which add automated analysis for finite scopes via model finding and model checking.

The Alloy language allows modellers to describe a system via sets and relations, but lacks explicit modelling constructs for describing a transition system. It is possible to parametrize all dynamic (changing) elements of the transition system by a “state” or “time” parameter but this practice is adhoc and makes it more difficult to read the Alloy models. Electrum [5] and DynAlloy [6], [7] are both extensions to Alloy[1] for describing transition systems. Electrum uses linear temporal logic (LTL) [9] to describe the traces of the transition system and DynAlloy describes transitions using an action language closer to programming languages. Neither provides any explicit constructs for decomposing system behaviour into hierarchical states or processes.

Dash [10], [11] is another extension to Alloy that uses the paradigm of hierarchical, concurrent states from Statecharts [12] for modelling transition systems. In Dash, a modeller creates a hierarchy of named control states and an explicit transition construct has a source and destination control state, a guard, and actions. Variables declared within the state hierarchy are considered dynamic. The guard and actions of a transition are described in Alloy and can refer to current (unprimed) and next (primed) values of the variables. Using the common control modelling paradigm of Statecharts (also used in UML state machines [13]) makes declarative behavioural modelling more approachable to users. Dash is translated to Alloy (without extensions) for analysis, thus the Alloy Analyzer can be used to check Dash models. However, Dash lacks a way to describe processes, such as a set of clients and a set of receivers, which are replicated copies of concurrent (AND) states.

Modelling in TLA+ is also based on sets and relations. Its extension, PlusCal [14], adds a process language construct. A model with replicated processes is called a parametrized system because it is parametrized by the number of copies the process. The number of copies of each process in the model analyzed can be chosen at the analysis stage. The combination of abstract data modelling plus the process construct is very valuable for modelling distributed systems as witnessed by the many case studies that use TLA+ or PlusCal (*e.g.,* [15], [16]). However, processes in PlusCal must be at the top-level of the model description and cannot be nested, and PlusCal lacks constructs for more interesting control state modelling as are found in Dash. Other declarative languages such as B [17], Event-B [18], and ASMs [19] do not have any constructs for explicitly modelling concurrent components.

PROMELA [20] is a lower-level model checking language used as the input language to the Spin model checker. PROMELA includes processes as a construct at the top-level of the model description and has synchronous and asynchronous channels for communication. However, its constructs for describing changes to data are limited. Zave [21] compares Alloy and PROMELA, and notes that an advantage of PROMELA is its process construct, which is lacking in Alloy. For example, in Zave’s model of the Chord protocol in Alloy [22] (which we use as an example later in our paper) all dynamic elements of the system must be parametrized by both a state and a process identifier. All transitions of the Chord transition system must be described from a global per-

[1] In its latest release, Alloy has incorporated Electrum into its language [8].

spective rather than a process-local perspective. This required parametrization and the global point of view both make it difficult to read the Alloy model and understand the details of a local process' behaviour.

Lacking is a formal modelling language that has explicit constructs for declarative modelling, hierarchical states, and replicated concurrent (possibly nested) processes. Such a language would provide a flexible means of modelling protocols, distributed systems, and many other domains, such as tracked-based traffic control systems [23], abstractly. This language would enable quick feedback on models for exploring options and finding bugs early in system development.

In this paper, we explore the syntax and semantics of an extension to Dash that includes replicated concurrent, nested states, which can model processes. We show how explicit constructs for replicated states make it possible to create models that are nicely decomposed because all aspects of a process can be declared and described together. The processes can communicate via buffers or exchange information through global variables and events. A key contribution of our novel approach is that a modeller can abstractly model the topology of the processes (ring, list, *etc.*) through constraints on the set indexing the process. **Dash+** is the name of our extension to Dash.

Our paper is organized as follows: Section II describes Alloy and Dash as background for Dash+. Section III presents Dash+. We use the Chord example to highlight the decomposition/localization effect achieved in Dash+. In Section IV, we discuss our proposed semantics for Dash+. Section V describes issues with respect to the analysis of Dash+ models. We conclude the paper with related work in Section VI.

II. Background

In this section, we briefly present Alloy and Dash as background for the rest of the paper.

A. Alloy

In Alloy [3], a model consists of sets and relations, and formulas in relational logic plus transitive closure that describe constraints on the sets and relations. A signature consists of a set declaration plus the declaration of fields (relations on the set) giving the language an object-oriented style. The sets can be arranged in a hierarchical manner through subsets and subset extensions (mutually exclusive subsets). Using the Alloy Analyzer, instances that consist of values for the sets and relations that satisfy the formulas are found using an underlying solving engine via a **`run`** command. Counterexamples to formulas can also be found using a **`check`** command. Alloy and its toolset have been used to model a variety of problems such as network protocols [22], security [24], and train control [25]. Alloy does not have any built-in constructs for dynamic system behaviour. Typically, a "state" set is created with predicates describing transitions as changes to before and after values of the state. The state set is then required to be an ordered set (a trace) where the ordering operation is that one transition must be satisfied in each step of the order.

B. Dash

Dash is an extension to Alloy that adds constructs for named control states in a hierarchical and concurrent arrangement for modelling transition systems. As in Statecharts, an AND-state describes concurrent behaviour and an OR-state is decomposed hierarchical behaviour. AND- and OR-states can be arbitrarily nested. The states are related by an explicit transition construct. The optional elements of a transition are (taken from [11]):

```
trans trans_label {
  from <src_state>
  on <trigger_event>
  when <guard_condition_in_Alloy>
  goto <dest_state>
  do <action_in_Alloy>
  send <generated_event>
}
```

A transition can include the source and destination control states for the transition (**`from`**, **`goto`**); events that trigger the transition (**`on`**) and that are generated by the transition (**`send`**); and actions (**`do`**) and guard conditions (**`when`**), which are expressed declaratively in Alloy to change the dynamic variables of the model. Dash allows for some attributes of the transition to be omitted and suitable defaults are chosen based on the transition's textual location within the model (*i.e.,* when a transition is declared within a state, it default source control state is its enclosing state). Dash has syntactic sugar called transition comprehension for declaring many transitions in one statement (such as every state transitions to an error state on some event).

Dynamic variables can be declared within any state, and a primed variable within a condition or action represent the next value of a dynamic variable after a transition. Communication in Dash is global via shared variables and events. However, using the name spaces provided by the state hierarchy it is possible to do directed communication by using unique names. Declaration of signatures and other regular parts of an Alloy model can be included in Dash outside of the state construct.

An extract from a Dash model of the game musical chairs is presented in Listing 1. The top-level state is a concurrent state (but there can be only one top-level state). Inside the top-level state, the transition system is decomposed into OR-states for the phases of the game: `Starting`, `Walking`, `Sitting`, and `End`. One example transition is shown that goes from `Walking` to `Sitting`. This example shows the very declarative nature of the transition's actions: an action requires that all seats are taken by one player, but we do not need to specify the many options for which seat is taken by which player (as would have to be done in an SMV model [26]).

A semantics was chosen for Dash in keeping with the common semantics for Statecharts where a big-step, which is the result of environmental inputs, can consist of multiple transitions arranged in a sequential order of small-steps. Considering the taxonomy of Statecharts semantic choices created

```
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 { ... }

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

Listing 1. Part of Dash+ Model for Musical Chairs [10]

by Esmaeilsabzali *et al.* [27], Dash allows one transition per small-step with cascading events and variable changes effective immediately after a small-step. Only one transition per concurrent region can be taken. These semantic choices ensure that all big-steps terminate and that the cause-effect behaviour is understandable in a strictly forward direction (an event must be generated before it can trigger another transition). With respect to variable changes in a transition, Dash unifies the declarative and operational modelling paradigms: if a variable is explicitly mentioned in the action of the transition, then the action constrains the variable, but if the variable is not mentioned in the action then it keeps its value from the previous snapshot.

Dash is translated to Alloy for analysis. In this translation, the state hierarchy is represented succinctly using Alloy subset extensions. Transition behaviour is captured in predicates for the pre-condition, post-condition and semantics of the transition, where the semantics predicate ensures the global requirements that only one transition is taken per concurrent region in a big-step, *etc.* Dynamic variables are translated into relations on a set of states. Any method for checking properties of the transition system such as inductive invariant checking, bounded model checking [28], [29] or transitive-closure-based model checking [26] can be used for analysis.

III. Dash+

Our goal in creating Dash+ is to support replicated AND-states to Dash. Intuitively, replicated AND-states should run concurrently with each other and with non-replicated AND-states of the model. The value of having replicated AND-states is that many systems such as protocols and distributed systems consist of replicated components. It is common, for example, to have a set of clients and receivers. We would like to be able to describe such systems abstractly in a model without knowing the number of replicated processes in advance. We are not aware of a modelling language that supports all of the features of 1) abstract data modelling; 2) hierarchical and concurrent control states; and 3) replicated (and possibly nested) concurrent states. The challenges in adding replicated concurrent states to Dash are in ensuring that 1) they can be modelled seamlessly with the control state hierarchy and behave as if they had each been modelled as a concurrent state individually for big/small steps; 2) we can easily model arrangements of components in linear orders or rings, *etc.* ; 3) they can communicate with other replicated and non-replicated states through global/local variables, events and buffers; and 4) we seamlessly integrate with Alloy.

We present Dash+ via discussions of 1) parametrization; 2) communication; and 3) modularity in the following subsections. The Dash+ models that we mention are available at: https://github.com/WatForm/watform-models .

A. Parametrization

We want our models to be able to have multiple different replicated AND-states (*e.g.,* a set of clients and a set of servers). We also want to defer setting the number of each kind of replicated AND-state until analysis time (as is done for any set in Alloy). As a language design decision, we consider whether the replicated states should be explicitly or implicitly parametrized. Implicit parametrization has the advantage of avoiding the excessive syntax of needing to include that parameter for every locally declared dynamic variable. However, with implicit parametrization, the component would have no means of communicating with its sister component directly (because it cannot address the sister component) and could only broadcast a message. Our solution is to explicitly parametrize the AND-state declaration but implicitly parametrize the locally declared dynamic variables.

We use the Chord protocol [30] as an example because it has also been modelled in Alloy by Zave [22]. Chord is a peer-to-peer distributed hash table that assigns keys to nodes that are organized as a ring. Each node has a unique identifier and a successor and predecessor node. Fig. 1 is an illustration of a Chord system. An active node is a node that is a part of the ring in the Chord system. Each active node in the Chord system is shown as a green circle with an unique identifier in Fig. 1. The successor for a node is displayed with a blue arrow and its predecessor with a yellow arrow. It is possible for

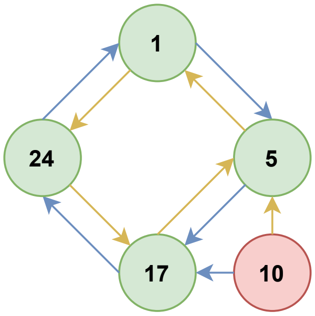

Fig. 1. Chord Nodes in a Ring (figure inspired by Zave [22])

```
 1  open util/ring[Node]
 2
 3  sig Node {}
 4  sig Succs {
 5    list: seq Node
 6  } { ... }
 7
 8  conc state NodeProc [Node] {
 9
10   succ: one Succs // list of successors
11   prdc: one Node
12   status: lone Status
13   saved: lone Node
14
15   env event Fail {}
16   env event Join {}
17
18   default state Active {
19    default state Stabilizing {
20     trans StabilizeFromSucc { ... }
21     trans StabilizeFromPrdc { ... }
22    }
23    state Rectifying {
24      trans Rectify { ... }
25    }
26    trans NodeFailure { ... }
27   }
28
29   state Failed {
30    // add this node to the ring
31    trans NodeJoin {
32     on Join
33     when status = Failed
34     do {
35      status' = Active
36      some otherNode: Node |
37       not (otherNode = this) &&
38       NodeProc[otherNode]/status=Active &&
39       between[otherNode, this,
40        NodeProc[otherNode]/succ.list[0]]&&
41       succ' = NodeProc[otherNode]/succ &&
42       prdc'= otherNode
43     }
44     goto Live
45    }
46   }
47   ...
48  }
```

Listing 2. Dash+ Model for Chord

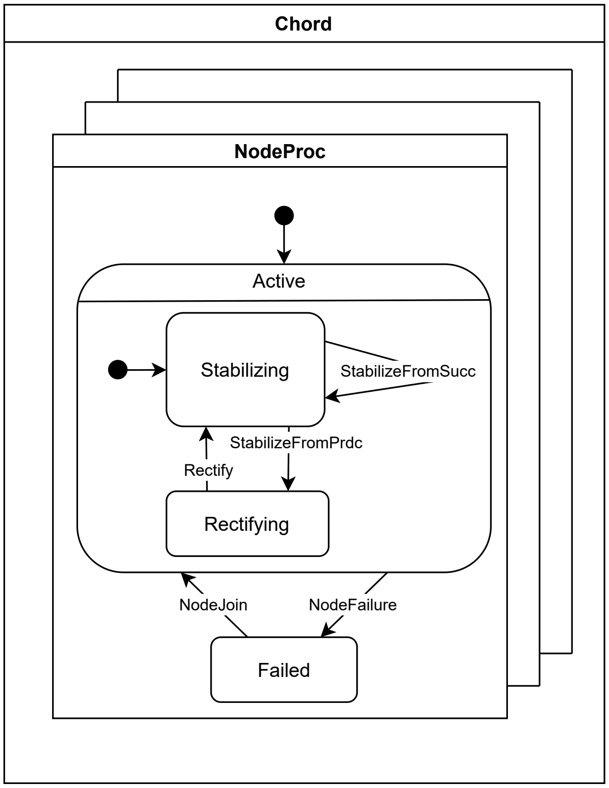

Fig. 2. State Hierarchy in the Dash+ Chord Model

a node to fail and have itself removed from the ring. When it rejoins the ring (as node 10 is doing in Fig. 1), it sets it successor and predecessor to point the node next and previous to it respectively. The successor and predecessor pointers for nodes 17 and 5 will be adjusted by stabilize and rectify actions within nodes 17 and 5 after node 10 joins the ring.

Listing 2 shows parts of our Dash+ model of the Chord protocol. Each replicated NodeProc state represents a node in the Chord system. Line 8 of Listing 2 shows the concurrent state NodeProc is parametrized by a set called Node. Any Alloy set can be used as the parameter set. This set is declared in a signature (on Line 3) outside of the state construct. The size (scope) of the Node set is set at analysis time in an Alloy run/check command as is normally done in Alloy.

By allowing the parameter to be any Alloy set, constraints on the parameter set can be written separately to choose a topology of communication for the replicated components. For example, the ordering library module in Alloy constrains a set to be a linear order and provides relations such as next and prev as operations on the elements of the order. Applying the ordering module to a set of identifiers for replicated states forces the replicated states to be arranged in a linear order with respect to each other, which governs their communication connections. The Chord protocol requires the nodes to be arranged in a ring. We plan to create a parametrized ring module in Alloy that is an extension of the linear ordering module in which the last element in the linear order points to the first element and vice versa. On line 1 of Listing 2, the Node set is constrained to be a ring. In another Dash+ example for a BitCounter, we use a linear order topology of replicated components to cascade the carry bit where each bit is one copy of a replicated AND-state.

In Listing 2, dynamic variables succ, prdc, status and saved are declared locally within the state (Lines 10– 13). The events Fail and Join come from the environment because they are declared using the keyword **env** (Lines 15– 16).

The steps of the Chord protocol are divided into labelled control states for the behaviour of each NodeProc, such as the states Active and Failed shown in Listing 2. Fig. 2 is an illustration of the state hierarchy in the Dash+ model for the replicated NodeProcs. Every NodeProc defaults to start in the Active state. This state has two substates within it: Stabilizing and Rectifying. Failure of a node will result in it exiting the Active state and any substate of the Active state on transition NodeFailure, and moving to the Failed state.

There is a transition named NodeJoin from the state Failed that reacts to the Join event when the status variable has the value Failed (guard). The source state of the transition NodeJoin is the state Failed because the transition is described with the Failed state and has no **from** part in the transition description. Within the **do** block of the transition NodeJoin is a list of Alloy formulas describing the change in behaviour that results from taking this transition in the model. The variable status is set to active in the step (because its primed version is used) and this node is fit into the ring between a different node (otherNode) and the other node's successor. We will discuss these actions in the next subsection, however note that explicit parameterization of NodeProc is used to refer its sister replicated components in the action. Finally, the destination state of the NodeJoin transition is the state Active.

We compare our Dash+ model with an extract from Zave's Alloy model of Chord [22] for the join operation shown in Listing 3. Zave's model follows a common style in Alloy modelling of transition systems where a predicate is used to model each transition system. The preconditions and postconditions are determined by comments. In Zave's model, "j" is the node joining the ring and is passed as argument to the predicate whereas in our Dash+ model, the operation of joining is happening to the current node when it receives the Join event. The previous and next states of the transition system are passed as arguments to the predicate and every dynamic variable (such as members) must be explicitly indexed by the state via the Alloy join ('.') operator. The set of all members of the ring is a dynamic variable of the state and changed directly rather than each node implicitly being a member of the ring when it is active and communicating with its neighbours. The succ and prdc elements map from states to nodes to values, whereas in Dash+ these elements are local to each process and do not require any of these parameters. Finally, because of the semantics of Dash, we do not have to model anything about the saved variable in the NodeJoin transition because it is not changed in this transition. Overall, Dash+ allows a **local** view for the node rather than thinking about the global state of the system for each transition.

```
pred Join [s, s': NetState, j: Node] {
-- PRECONDITIONS
   j ! in s.members
   some m: s.members |
      between [m, j, m.(s.succ).list[0]]
-- j queries m to get its successor list
-- POSTCONDITIONS
   && s'.time = next [s.time]
   && s'.members = s.members + j
   && s'.succ = s.succ + (j -> m.(s.succ))
   && s'.prdc = s.prdc + (j -> m)
   && s'.status = s.status
   && s'.saved = s.saved
}
```

Listing 3. Part of Zave's Alloy Model for Chord [22]

Dash+ allows nesting of replicated components. In one of our examples, we model a heating system with concurrent states for a controller, a furnace, and multiple rooms. The rooms are replicated components. On error or off events, the system exits these concurrent states and moves to error/off states. This model includes abstract data operations that use quantification over the rooms requesting heat. The feature of nesting replicated components is unique to Dash+ within the family of declarative languages.

B. Communication

Concurrent components of a model have to be allowed to communicate with each other. In Statecharts, communication is possible via global variable access and global events. While all communication is global, directed communication is possible by using particular named events. Dash helps aid in locality of communication through its namespaces – while everything is accessible globally, referencing a variable or an event in another component must be prefixed by the component's name.

In Dash+, a replicated component can receive global environmental events or send global events (*e.g.,* a request for a token) as these events are declared outside of the component. Internal access to its locally declared events/variables is written without parametrization. The only time parametrization is needed is for communication to/from sister components of this component. Examples of this use of parametrization can be seen on Lines 38– 41 in Listing 2. Using NodeProc [otherNode]/succ, we refer to the succ variable in the copy of NodeProc indexed by otherNode. Occasionally, it is useful to refer to the index of the current node via the keyword this. For example, NodeProc[this].next refers to the next node to this one in a linear order.

Asynchronous Buffers. Protocols and distributed systems often communicate via buffers (channels). When constructing the model, the size of these buffers may not be known and it is important that a buffer size limit not be fixed within the model. No built-in library in Alloy works for Dash+ buffers because a modeller may want two different buffers consisting

```
1  conc state Server [ServerID] {
2   requests : buf[ClientID]
3
4   default state Sending {
5    trans send {
6     // There is a request by a client
7     when !requests.isEmpty
8     do {
9      // Send an integer to the buffer of the
10     // client making the request
11     one x : Int |
12      Client[requests.first]/messages.Add[x]
13      // Clear that request
14     requests.delete[0]
15    }
16   }
17  }
18 }
19 conc state Client[ClientID] {
20  messages : buf[Int]
21  ...
22 }
```

Listing 4. Part of a Dash+ Model with Buffers

of elements of the same set to have different sizes. Therefore, we introduce syntax in Dash+ buffers as shown on line 2 of Listing 4. In this example, there are replicated `Server`s each with a buffer called `requests` of `ClientID`s and there are `Client`s each with a buffer called `messages` of **`Int`**s. When a Server has a client id in its buffer (line 7) it chooses an integer to add to that client id's buffer and then deletes this element from its buffer (lines 11– 14). There is no **`goto`** part of the transition because it uses the default of looping back to its source state. We are investigating whether to make all buffers first-in, first-out (FIFO) or to provide different access policies for buffers.

Synchronous Buffers. The original modelling languages for protocols, CSP [31] and CCS [32] included synchronous/handshake/rendezvous communication between components where the sender and receiver both take a transition on the shared event. The Statecharts model of communication, however, relies on a sequence of small steps that describe the cascading effect of an event triggering a transition, which may generate another event and trigger another transition. This form of communication has been well studied (*e.g.,* [27]) and relates to Berry's synchrony hypothesis [33], which assumes that the system can complete its response to an environmental event before the environment generates another input. In Dash+, we are creating a language that includes both replicated components and their interactions along with hierarchical state modelling, which means we have to combine the typical semantics of each in an intuitive manner. Looking at one big step in Dash+ as the system's response to the environment allows us to view a sequence of small steps in a combined way as a synchronous response, thus we do not need to support a special kind of synchronous buffer. If one component puts something in a buffer, another component can read from that buffer within the same big step.

C. *Modularity via Multiple Files*

An important aspect of modelling, even at a very abstract level, is decomposition into multiple files. Following Alloy and Dash, Dash+ has two methods of decomposing models:

- Existing and new Alloy modules provide packages of constraints on data. An example of how this is used in Dash+ is for the topologies of the replicated processes, *e.g.,* the ordering module, which orders the components and allows them to access their neighbours via functions.
- Dash decomposition is accomplished via file concatenation. Each kind of replicated component can be described in one file and the files can be concatenated to form a Dash module directly because Dash does not require a single root state.

IV. Semantics

The semantics of replicated concurrent components in Dash+ is that they behave as a set of concurrent states. Dash models are translated to Alloy for analysis [10]. The translation implements the semantics of hierarchical, concurrent states in big- and small-steps. For the analysis of Dash+ models, we plan to extend Dash's translation to Alloy. Rather than simply replicating the component a fixed number of times during translation, a key challenge in this translation is to not set the number of replicated components until analysis time. Similarly, we want to leave the size of the buffers to be set at analysis time and all buffers should not have to be the same size.

We are exploring how to build parametrization into the translation. All locally declared variables and events of a replicated component will become functions that are parametrized by the replicated component's identifier set in addition to a state set. Therefore, the number of copies of each dynamic variable included in the model relies on the cardinality of the replicated component's identifier set, which is decided at analysis time.

Control state and transition labels are translated to sets in Dash (rather than functions) so parametrization does not work directly for their translation. We have to link the cardinality of copies of these sets to the number of replicated states at analysis time. A simple method of solving this issue is to use Alloy's set cardinality operator and constrain the cardinality of the sets of state and transition labels to be equal to the cardinality of the replicated component's identifier set. However, using set cardinality is not efficient and increases the analysis time for Alloy models. A more efficient approach is to specify the scope for each set in the run/check command such that its scope is equal to the scope of its replicated process identifier set. But, it would be tedious to require this of the user when using the Alloy Analyzer on Dash+ models. Thus, we are integrating support for Dash+ directly into the Alloy Analyzer to hide the process of translating Dash+ to Alloy. Via this integration, we can hide the need to make these sets equivalent in scope and still use the more efficient method of setting the scopes in the command directly.

For buffers, we want to allow different buffers consisting of elements of the same set to have different sizes. Alloy's built-in **seq** module is close to our desired meaning for buffers (a function mapping an index to a position in a sequence). This module allows users to create buffers consisting of elements of a set and add/remove items to/from any index respectively. However, it requires all buffers consisting of elements of the same set to be the same size because the sequence module has only one index set. Once the scope for the index set has been specified for a buffer, every buffer will the size of the index set. Alleviating this issue requires the use of multiple index sets such that each buffer will have its own index set. We will allow the user to set the size of the buffer in a run/check command by buffer name (as in `run for 3 requests, 4 messages`). Typically in Alloy, a user cannot set the size of a relation directly (only sets) so our translator will create a buffer relation (with appropriate operations for FIFO or other access policies) and use the scopes set for each buffer in the command in Dash+ as the scopes of the index sets in the Alloy translation.

One important aspect of the semantics that needs to be explored further is the possibility of transitioning into or out of a substate of a replicated component. Transitioning out of a substate of a replicated component is the same as transitioning out of one component of a concurrent state, which is already supported in Dash. In this case, every component of the concurrent state is exited when the transition is taken. Transitioning into the substate of a replicated component is effectively creating a transition with multiple destination states. This case currently does not exist in Dash as a transition cannot have multiple destination states. However, the semantics of this case should be that the destination substate is entered in all replicated components. Transitions from a substate of one replicated component to a substate of another replicated component will be disallowed through a well-formedness condition.

V. Analysis

We plan to provide analysis support for Dash+ via the Alloy Analyzer. As was done with Dash, we will implement the semantics described in the previous section as a translator to Alloy.

For model checking, we want to be able to write properties that are requirements of all replicated components. For example, in the Chord model, an active node must contain a predecessor (`prdc`) node. Using regular Alloy quantification over the `Node` set parametrizing the replicated AND-state and the embedding of computation tree logic (CTL) [34] provided in [35], we can write:

```
assert alwaysOnePredecessor{
    ctl_mc
        [ag[all id: Node |
           NodeProc[id]/status=Active
         => one NodeProc[id]/prdc]]}
```

For the visualization of instances and counterexamples, using Dash+, the instance will include an identifier for each process along with information on transitions taken (because we have named transitions) and the current control state. This extra information will aid in the debugging process.

Replicated components will cause larger state spaces for analysis. ALDB [36] provides an initial method for debugging Dash+ models. ALDB is a debugger for Alloy models of transition systems. It allows a modeller to step through a transition system and write simple bounded model checking properties ('until'). No special support is needed in ALDB for either Dash or Dash+ models, but we are exploring how a graphical illustration of the named state identifiers can help a user in understanding the model during simulation.

To handle large state spaces, we will develop additional significance axioms [26]. Significance axioms in Alloy determine a scope for the transition system that covers an important subset of its behaviours. For example, a significance axiom can require the instances checked to include paths that cover all declared transitions. A significance axiom can require that all states in the instance are reachable from an initial state to avoid spurious instances. In Serna [10], significance axioms were created for Dash that require that every control state is reachable and all big steps are complete. For Dash+, a new significance axiom could require that at least one copy of every replicated component participates in a path of the transition system.

By exposing the replicated components syntactically, there may be opportunities to apply symmetry breaking to reduce the size of the state space. For example, partial order reduction [37] reduces the number of interleavings of transitions to consider in state space search. In addition, results from model checking parametrized systems [38] may be applicable for generalizing results from checking a finite number of components. For example, Bozga *et al.* [39] and Ezparza *et al.* [40] describe methods for efficiently proving safety properties of models with replicated concurrent components using invariants.

VI. Related Work

Wallace [41] modelled processes in Alloy by defining a model that changes its global state using transitions with pre- and post-conditions. Each dynamic variable of a process is modelled as a function from the global state and process identifier to the value. Zave [22] used a similar modelling method in Alloy for nodes in the Chord ring. This extra parametrization is tedious and makes it difficult to understand the model from the point of view of one process. The models of Wallace [41] and Zave [22] did not use buffers. Dash implicitly models time (or a step) for the user and with our contributions in Dash+, processes with buffers can be modelled explicitly.

Electrum [5] is an extension to Alloy that includes the keyword "var" as syntax to denote the declaration of dynamic elements of a model. Primed uses of the variable denote values of the variable in the next step of the transition system.

It uses LTL to describe both the transition system (traces) and the properties to check of the model. There are also constructs to describe individual transitions called actions [42]. DynAlloy [6], [7] is an extension of Alloy that enables users to define a system configuration (an initial state of the system) and reconfigure the system (change its state) using an action. The actions contain a pre-condition and post-condition to describe changes to the system after the action and are strongly influenced by programming language constructs. Both of these extensions lack state hierarchy, replicated concurrent states, and communication using buffers, which Dash+ provides.

PlusCal [14] is an extension of TLA+ [4] that includes explicit language constructs for replicated processes. Processes in PlusCal are explicitly indexed and communicate via shared variables. There is no explicit buffer construct. Replicated processes in PlusCal run concurrently via interleaving. A transition from one process is taken in a step. Properties of PlusCal models can be checked via a translation to TLA+ and using a model checker or theorem prover for TLA+. Any expression in TLA+ can be used in the triggers and actions of transitions making it possible to model data very abstractly, but PlusCal lacks control state hierarchy as is found in Dash+. All processes in PlusCal must be at the top-level, whereas in Dash+ replicated AND-states can be arbitrarily nested.

Spin [20] is a model-checking checking tool that analyzes models written in the modelling language PROMELA. In PROMELA, users can declare concurrent replicated processes and the replicated components can communicate with each other using global variables and/or buffered/rendezvous channels. However, PROMELA has limited datatypes and data operations because it focuses mainly on the communication and synchronization aspects of a model. PROMELA also lacks hierarchical state modelling.

UML state machines [13] support hierarchical and concurrent labelled control state hierarchy. Through the use of object modelling, UML supports replicated concurrent components. Using OCL [43], pre- and post- conditions and invariants can be included in the UML model. However, UML models lack the level of abstraction for data descriptions that a declarative language such as Dash+ or Alloy can provide. By providing language constructs that fully integrate abstract data descriptions with control state modelling paradigms including replicated components, Dash+ can be used for data-oriented and control-oriented modelling.

VII. Conclusion

We have presented Dash+, an extension of Dash and Alloy, that provides language constructs for describing a model with replicated concurrent components. Dash+ allows communication among these components and between these components and the rest of the model. This communication can be global or directed based on a particular topological arrangement of the components and may be buffered or not. Dash+ does not extend the expressiveness of Alloy; it adds explicit language constructs for convenience in describing transition systems. Dash+ aims to be a flexible and abstract modelling language for transition systems that combines abstract data, hierarchical control states, and replicated components.

Dash+ can have multiple sets of replicated components in the model and these replicated components can be at any level in the control state hierarchy, which is a novel and powerful modelling feature. A key insight in Dash+ is that we can use a regular Alloy set to describe the topology of the replicated components. This generality allows users to arrange the replicated components in common (and uncommon) communication structures (*e.g.,* rings) using regular Alloy constraints. The elegance and abstractness of separating the modelling of the behaviour of a replicated component and the specification of the topology of the replicated components is unique to Dash+.

In addition, Dash+ provides an explicit construct for buffered communication to allow buffers consisting of elements of the same set to have different sizes. Combining the locality of replicated components with buffered communication and the simplicity of using an existing Alloy set to define the topology provides important new features to Dash.

We are working on implementing Dash+ as a translation to Alloy that implements the semantics we have described in this paper and is directly integrated within the Alloy Analyzer. Users will be able to toggle between a "Dash+" view of the model and an "Alloy" view of the model. We are continuing to investigate optimizations for analysis and case studies. Once we have support for Dash+ implemented, we plan to investigate common modelling patterns for replicated AND-states. For example, there are common methods for handling failures in replicated components in protocols and we may be able to provide a Dash+ library that provides models for these common patterns.

Acknowledgments

We thank Jose Serna (the creator of Dash) for discussions regarding the future of Dash. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] J. M. Spivey, *Z Notation - a reference manual*, 2nd ed. Prentice Hall, 1992.

[2] C. B. Jones, *Systematic software development using VDM*, 2nd ed. Prentice Hall, 1991.

[3] D. Jackson, *Software abstractions: logic, language, and analysis*, 2nd ed. Cambridge, Mass: MIT Press, 2012.

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

[5] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The Electrum analyzer: Model checking relational first-order temporal specifications," in *ASE*, 2018, pp. 884–887.

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

[7] G. Regis, C. Cornejo, S. Gutiérrez Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J. P. Galeotti, and M. Frias, "DynAlloy analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour," in *FSE*. ACM, 2017, pp. 969–973.

[8] "Alloytools/org.alloytools.alloy," https://github.com/AlloyTools/org.alloytools.alloy, 2021, [Online; accessed Jun 28, 2021].

[9] A. Pnueli, "The temporal logic of programs," in *Symposium on Foundations of Computer Science (FOCS)*. IEEE Comp. Soc., 1977, pp. 46—-57.

[10] J. Serna, "Dash: Declarative behavioural modelling in Alloy," MMath thesis, Univ. of Waterloo, Cheriton School of Comp. Sci., 2019.

[11] J. Serna, N. A. Day, and S. Farheen, "Dash: A new language for declarative behavioural requirements with control state hierarchy," in *MODRE Workshop @ RE*. IEEE Comp. Soc., 2017, pp. 64–68.

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

[13] "OMG unified modeling language," http://www.omg.org/spec/UML/2.5/PDF/, 2015, [Online; accessed Jul 4, 2021].

[14] L. Lamport, *A PlusCal User's Manual (P-Syntax)*, version 1.8 ed., August 2018.

[15] C. Newcombe, "Why Amazon chose TLA$^+$," in *ABZ*, 2014, pp. 25–39.

[16] H. Wayne, "List of TLA+ examples," https://www.hillelwayne.com/list-of-tla-examples/, 2021, [Online; accessed June 28, 2021].

[17] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[18] ——, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[19] E. Börger and R. F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[20] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[21] P. Zave, "A practical comparison of Alloy and Spin," *Formal Aspects in Computing*, vol. 27, no. 2, pp. 239–253, 2015.

[22] ——, "Reasoning about identifier spaces: How to make chord correct," *IEEE Trans. on Software Engineering*, vol. 43, no. 12, pp. 1144–1156, 2017.

[23] M. Bagheri, M. Sirjani, E. Khamespanah, N. Khakpour, I. Akkaya, A. Movaghar, and E. Lee, "Coordinated actor model of self-adaptive track-based traffic control systems," *Journal of Systems and Software*, vol. 143, 05 2018.

[24] E. Kang, S. Adepu, D. Jackson, and A. P. Mathur, "Model-based security analysis of a water treatment system," in *2016 IEEE/ACM 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. IEEE, 2016, pp. 22–28.

[25] A. Svendsen, B. Møller-Pedersen, Ø. Haugen, J. Endresen, and E. Carlson, "Formalizing train control language: automating analysis of train stations," in *Comprail*, 2010, pp. 245–256.

[26] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking in Alloy," *SoSym*, vol. 19, pp. 721–740, 2020.

[27] 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.

[28] A. Cunha, "Bounded model checking of temporal formulas with Alloy," in *ABZ*. Springer Berlin Heidelberg, 2014, pp. 303–308.

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

[30] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[31] T. Hoare, *Communicating Sequential Processes*, ser. International Series in Computer Science. Prentice Hall, 1985.

[32] R. Milner, *Communication and Concurrency*, ser. International Series in Computer Science. Prentice Hall, 1989.

[33] G. Berry and G. Gonthier, "The esterel synchronous programming language: design, semantics, implementation," *Sci. of Comp. Prog.*, vol. 19, no. 2, pp. 87–152, 1992.

[34] E. Clarke and E. Emerson, "Design and synthesis of synchronisation skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, ser. LNCS, vol. 131. Springer, 1981, pp. 52—-71.

[35] A. Vakili and N. A. Day, "Temporal logic model checking in Alloy," in *ABZ*, ser. LNCS, vol. 7316. Springer, Jun. 2012, pp. 150–163.

[36] A. Dureja, A. Keerthi, A. Liang, P. Zhang, and N. A. Day, "ALDB: Debugging Alloy models of behavioural requirements," in *MODRE Workshop @ RE*. IEEE, 2020, pp. 21–30.

[37] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, vol. 2, no. 2, pp. 149–164, 1993.

[38] P. A. Abdulla, A. P. Sistla, and M. Talupur, "Model checking parameterized systems," in *Handbook of Model Checking*. Springer, 2018, ch. 21, pp. 685–725.

[39] M. Bozga, J. Esparza, R. Iosif, J. Sifakis, and C. Welzel, "Structural invariants for the verification of systems with parameterized architectures," in *TACAS*, ser. LNCS, vol. 12078, 2020, pp. 228–246.

[40] J. Esparza, M. Raskin, and C. Welzel, "Computing parameterized invariants of parameterized petri nets," 2021.

[41] C. Wallace, "Using Alloy in process modelling," *Information & Software Technology*, vol. 45, no. 15, pp. 1031—-1043, 2003.

[42] J. Brunel, D. Chemouil, A. Cunha, T. Hujsa, N. Macedo, and J. Tawa, "Proposition of an Action Layer for Electrum," in *ABZ*, ser. LNCS, no. 10817. Springer, 2018, pp. 2–6.

[43] "OMG object constraint specification (OCL) specification," http://www.omg.org/spec/OCL/2.4/PDF, 2014, [Online; accessed Jul 4, 2021].