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

by

Tamjid Hossain

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, 2022

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 behavior (replicated processes) running in parallel as is found in languages such as PlusCal and Promela.

We propose extensions 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 replicated concurrent processes. The processes can communicate via buffers or exchange information through variables and events. The key contributions of our novel approach are:

1) Replicated and non-replicated components can be nested arbitrarily at any level in the state hierarchy

2) Replicated components can exchange information directly without resorting to global variables as is the case in PlusCal and Promela

3) A modeller can abstractly model the topology of the processes (ring, list, etc.) through constraints on the set indexing the processes

4) Buffers can be used to facilitate communication between replicated components

Dash+ stays consistent with the semantics of Dash and uses the notion of big steps and small steps to describe changes in the system. The semantics are implemented in a translation to Alloy in a way that accommodates the following model checking options: traces-based model checking, transitive closure-based model checking (TCMC), and Electrum.

Our implementation is fully integrated into the Alloy Analyzer. This thesis presents case studies to demonstrate the features of Dash+ in modelling systems with concurrent processes and the benefits that Dash+ offers over existing languages. We check for properties in each of the models in the case studies to demonstrate how different model checking options can be used.

Acknowledgements

I would like to thank my supervisor, Prof. Nancy A. Day for her constant support throughout my Master's program. We completed this thesis throughout the pandemic, and Prof. Nancy A. Day's constant support, caring nature, and endless patience greatly assisted in helping complete this thesis. I will forever be thankful and grateful to her for her endless support.

I would also like to thank my second readers, Prof. Richard Trefler and Prof. Derek Rayside for providing insightful comments in a short period of time.

I would also like to thank my colleagues Elias Eid, Amin Bandali, Aditya Shankar Narayanan, and the original creator of Dash, Jose Serna with whom I have engaged in fruitful conversations with.

Finally, I would like to thank my family for constantly supporting me throughout my life and trying their best to help me achieve my dreams. None of this would be possible had it not been for them rooting for me every step of the way.

Dedication

I would like to dedicate my work to my parents, my brother, and my close friends Asif, Hadi, Rehan, etc. They mean the world to me, and their endless encouragement has kept me going. My mom has sacrificed her entire career to raise me and my brother in her own hands, and this work is a testament to her sacrifice. I hope I have made you proud, ammu.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the world becomes intertwined with technology, it is critical to have hardware and software systems that are safe and reliable. This fact is of paramount importance with safety-critical software systems where failure is not acceptable. One method of trying to assure that a software system will function correctly is through the use of a validation technique called model checking [13]. Temporal logic model checking aims to use models that describe a complex system and ensure that vulnerabilities and design flaws are discovered early in the development process.

A model is an abstraction of a system that is comprised of its essential elements. It describes the features that are critical in ensuring correct functionality while ignoring irrelevant details. A model can be defined to provide a human-understandable description of a system or be presented in a form that can be mechanically analyzed. This step is taken during the requirements phase of the software development cycle and assists software architects in discovering costly design flaws that would have transitioned into the implementation phase [32].

A model can be constructed using formal modelling languages such as Alloy [27] [26], Promela [24], TLA+ [34], etc. Models can be divided into two sub-categories: structural and behavioral. A structural model describes the relations between elements in a model, and a behavioral model describes how a system changes over time.

Alloy is a declarative language for describing models. It uses sets, relations, functions, predicates, and constraints between relations to describe abstract models. Given the abstract nature of an Alloy model, it is useful for receiving feedback early in the development process. The feedback is received through the Alloy Analyzer which translates Alloy constraints into Boolean constraints and solves them using SAT solvers. However, Alloy lacks explicit constructs for defining behavioral models but this issue has been addressed using the Dash declarative language.

Dash [41] is an extension of Alloy with explicit constructs for modelling transition systems and uses a control state hierarchy inspired by Statecharts [22]. Statecharts is a control-oriented behavioral modelling language that can describe reactive systems. Control-oriented models describe complex behaviors for changes in a system through transitions. A reactive system is event-driven and reacts to internal or external stimuli. In Dash, modelers can define explicitly named control states arranged in a hierarchy with control states being related using transitions. The guards and actions of a transition are described using Alloy expressions. This gives Dash the ability to create reactive systems while supporting rich data types for more complex operations. Once a Dash model has been designed, the behavioral properties of the model can be checked using the Alloy Analyzer after it has been converted to an Alloy model.

One feature that Dash lacks is process constructs and communication between replicated processes. This feature is supported by languages such as PlusCal [35] and Promela. PlusCal is an extension of the TLA+ [34] modelling language. PlusCal enables the creation of multiple replicated processes that communicate with each other using buffers. Similarly, Promela can be used to define models with replicated processes that communicate through both synchronous and asynchronous buffers. Zave [48] compared the use of Alloy and Promela for modelling the CHORD protocol. She noted that safety properties are easier to write in Alloy (since CHORD properties are graph properties) and that the startup time for Alloy is less than Promela (since a modeller will need to learn C to write properties in Promela). Dash+ seeks to be a language that has features of both Alloy and Promela for modelling protocols and distributed systems.

Our work aims to extend Dash with constructs for creating processes that can communicate with each other. We present Dash+, an extension to Dash that allows modellers to model reactive systems with replicated processes that can communicate by accessing variables in sister processes or using by using buffers. With the formal process construct in Dash+, a modeller can describe behavior from one process' point of view rather than taking a global view of describing the behavior of all processes together as is done in modelling CHORD in Alloy [47]. In Dash+, we can create an arbitrary nesting of replicated and non-replicated processes that run concurrently. This feature is not currently supported by any formal modelling languages with processes constructs to our knowledge.

The importance of modelling languages with process constructs can be seen in the case studies for PlusCal and Promela (e.g [39] [30] [14]). They are essential in modelling distributed systems, protocols, air-traffic control systems, etc. in which multiple actors communicate with each other and run concurrently together. An example of a model with multiple actors communicating is a client-receiver model with a set of clients and receivers running concurrently (and communicating with each other). Such a model is a parameterized model since it is parameterized by the number of copies of the process. In Dash+, the notion of replicated processes is achieved through parameterizing concurrent states. The modeller can decide on an upper bound for the number of copies each replicated concurrent state. This has a beneficial side effect of enabling a replicated process to communicate with a sister process by referring to it by using a parameter and using constraints on the parameter set to arrange the processes in various topologies (ring, list, etc.).

Communication between processes is made easier through the use of buffers. In Dash+, the size of a buffer can be specified in a command. Each buffer in the model can have its own unique size. Replicated processes can access the buffer of a sister process and add or remove items in the buffer as defined by the modeler.

We have integrated Dash+ into the Alloy Analyzer (version 6). A modeller can define a Dash+ model within the Alloy Analyzer, have it internally converted to an Alloy model, and display any instances or counterexamples. Modellers may also choose from one of three model checking options: traces-based model checking, transitive-closure-based model checking (TCMC), or Electrum.

1.1 Contributions

The contribution of this thesis is:

- Dash+ has constructs for both replicated and non-replicated concurrent components.
- Replicated and non-replicated concurrent processes in Dash+ can be arbitrarily nested.
- In Dash+, the arrangements of processes in a topology can be specified independently of the process using constraints on the parameter of the process.
- Dash+ has constructs for concurrent processes to communicate asynchronously using buffers. Sister components in Dash+ can directly communicate with each other by accessing variables.
- Dash+ expands on the model checking options in Dash by providing three model checking options: traces-based model checking, transitive-closure based model checking (TCMC) and Electrum.

- Dash+ has been integrated with the Alloy Analyzer.
- We have conducted case studies to demonstrate the features in the Dash+.

The fundamentals of the syntax and semantics of Dash+ have been published in [25].

1.2 Validation

We evaluate Dash+ using case studies that exploit features of Dash+ to demonstrate its ability to model transition systems with processes running concurrently. The case studies demonstrate the following features in Dash+:

- Replicated and non-replicated components running concurrently
- Direct communication between sister components in a replicated component using variables and events
- Communication through the use of buffers
- Arbitrary nesting of replicated and non-replicated components
- Ease of writing graph properties in models that use a graph structure
- Arranging processes in different structures such as ring, tree, etc.
- Model checking each of the models with the model checking options available in Dash+

1.3 Thesis Outline

Chapter 2 provides background on the Alloy language and the syntax and semantics of Dash. Chapter 3 presents the syntax and semantics of Dash+ and how a model with replicated AND-states can be created in Dash+. Chapter 4 describes the translation of a Dash+ model to Alloy and focuses on the translation of the state hierarchy and buffers. Chapter 5 presents how each temporal logic model checking method is connected to a Dash+ model. Chapter 6 presents case studies that demonstrate the features of Dash+ and outlines how each case study demonstrates specific features in Dash+. Chapter 7 compares Dash+ with closely related modelling languages and the advantages that Dash+ offers over them. Chapter 8 provides the conclusion and points to any future work for improving Dash+.

Chapter 2

Background

This chapter presents a brief background on Alloy, and an extension to Alloy called Dash. We will briefly discuss the syntax and semantics of Dash.

2.1 Alloy

Alloy is a modeling language based on relational logic that consists of sets, relations and transitive closure [26]. All structures in Alloy are based on relations, and a set in Alloy is considered a unary relation. Sets are declared using signatures and each set consists of atoms (or elements) that are indivisible, immutable and uninterpreted. The sets can be arranged in a hierarchical manner through subsets and subset extensions (mutually exclusive subsets). We can additionally declare abstract signatures which only have elements belonging to its extensions. Figure 2.1 shows an Alloy model with three signatures: `A`, `B` and `C`.

```
sig A {} // set of atoms called A
sig B {}
sig C extends B {} // subset of B
```

Figure 2.1: Signatures in Alloy

A signature can be comprised of fields that declare relations. A relation is a structure that relates atoms, and fields are declared with constraints that limit the multiplicity of the relations. The multiplicity constraints that are available in Alloy are: `lone`, `one`, `some` and `set`. A `lone` constraint means that a relation can map each domain element to one or zero elements; a `set` constraint means that a relation can map each domain element to any number of elements in a set; a `one` constraint means that a relation can map each domain element to exactly one element; a `some` constraint means that a relation can map each domain element to one or more elements. If a multiplicity constraint is not stated, then a relation will be automatically specified as a singleton set (`one` multiplicity). In Figure 2.2, the signature `D` has two relations: `f0` and `f1`. The relation `f0` is a mapping from an element in the signature `D` to any number of elements in the signature `A`; the relation `f1` is a mapping from every element in the signature D to exactly one element in the signature `C`.

```
sig D {
  // Fields
  f0: set A
  f1: one C
} {
  // Constraints
}
```

Figure 2.2: Fields in a Signature

Constraints in Alloy are written as Alloy expressions and use constants and set operators (union, intersection, etc.). These expressions either evaluate to a boolean value or result in a relation. Constraints can be defined in an optional block within a signature declaration or inside the body of a fact, predicate or function.

A fact consists of a collection of constraints that are always expected to hold (it should always evaluate to true) and is declared using the `fact` keyword. A fact may have a unique mnemonic name and any number of facts can be used in a model. Figure 2.3 shows a fact named `factOne` declared in an Alloy model with a constraint specifying that the relation f1 relates every element in set `D` in Figure 2.1 to one or more elements in the set `C`.

A predicate is a named set of constraints that takes in arguments. Since it is named, we can reuse the constraints for different contexts. Predicates declared with arguments must have instantiating expressions, and this gives it the versatility to be used in varying contexts. A function provides a template for an expression that takes in arguments and a

```
// fact called factOne
fact factOne {
  // Constraint
  // For all elements in the set D, the relation f1 must have
     one or more elements
  all d: D | some d.f1
}
```

Figure 2.3: Fact in Alloy

declaration expression for the result. A function is declared using the `fun` keyword, and a predicate is defined using the `pred` keyword. The predicate in Figure 2.4 takes in two sets: `a` and `b` as arguments; the function in Figure 2.4 takes one argument and returns an element from the set B (`one B`).

```
// predicate with two arguments 'a' and 'b'
pred p [a: A, b: B] {
  // Constraints
}

// function with an argument 'a'
// The resulting expression is an element from the set B
fun f [a: A] : one B  {
  // Constraints
}
```

Figure 2.4: Predicates and Functions in Alloy

An Alloy model can be analyzed by checking an assertion or running a predicate. An assertion is a constraint that is valid for all possible cases. A `check` command checks for instances in which an assertion does not hold. If an assertion fails to hold for a particular instance, then a counterexample is produced. The `run` command looks for instances that satisfy the constraints in a predicate. It should be noted that the logic of Alloy is undecidable. We cannot state with complete certainty that an assertion is valid. Hence, the size of instances considered is limited with a scope. The analyzer will check for every instance that is only within the specified scope.

The Alloy language has various common packages of constraints called modules that can be implemented into a model. For example, the commonly used ordering module can be used on a signature in an Alloy model to arrange the elements in the signature in a linear order.

2.2 Dash

Dash is an extension to Alloy for modelling transition systems [41]. It adds constructs for transitions and named control states in a hierarchical and concurrent arrangement. As in Statecharts [22], an AND state describes concurrent behaviour and an OR-state is decomposed hierarchical behaviour. AND- and OR-states can be arbitrarily nested. An AND-state is created using the `conc state` keyword. OR-states are created using the `state` keyword. A default state is specified using the `default` keyword. AND- and OR-states can include relations that change dynamically throughout the execution of the model, which we call **dynamic variables** or just **variables**. Variables are declared as relations with their respective multiplicity constraints. Events can be declared using the `event` keyword within the body of an AND-state. They can be either an internal event or an environmental event. Environmental events must be specified using the `env event` keyword. Figure 2.5 has a Dash model with an AND-state `C` that has two variables (`v0` and `v1`) and events (`E0` and `E1`). `C` has a basic state `S0` and an OR-state `S1` with a nested basic state `S2`.

A transition is created using the `trans` keyword. It can be named and has the following construct:

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

It includes the source (`from`) and destination (`goto`) states for the transition; events that trigger a transition (`on`) and internal events generated by a transition (`send`); the guard condition (`when`) and actions (`do`). The guard condition and actions are specified using Alloy expressions. The Alloy expressions in a guard condition must be satisfied for a transition to take place. Actions specify how the system (the variable values) will change as a result of a transition. The value of a variable is referenced by adding a prime to

```
//AND-state named C
conc state C {
  v0: lone A // variable
  v1: one A

  event E0 // internal event
  env event E1 // environmental event

  default state S0 { // default basic state for C
    trans T0{ // transition
      on E1
      when one v0
      do v1' = v0
      goto S1
      send E0
    }
  }
  state S1 { // OR-state
    default state S2 {} // nested basic state
  }
}
```

Figure 2.5: Example Dash Model

the variable's name. 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, its default source control state is its enclosing state).

Informally, the transition `T0` in Figure 2.5 can only take place when the system is the state `S0` (`on S0`), the environment event `E1` has been generated (`on E1`) and the variable `v0` is a mapping to a singleton set (`when one V0`). These are the preconditions for the transition. It is not necessary for us to specify the source state since this transition is declared inside the state `S0`. Once the transition takes place, the value of the variable `v1` is changed (`do v1' = v0`). The AND-state `C` will transition to the state `S1` (`goto S1`) and will generate the internal event `E0` (`send E0`).

The initial constraints of the transition system can be described using the `init` keyword. All the initial constraints are included in the body of an `init` statement using

Alloy expressions. Each AND-state can have initial constraints to describe the state of the system upon initialization. The initial constraints in Figure 2.6 specify that the variable `v0` in the AND-state `C` should not contain any values; the variable `v1` in the AND-state `C` should contain exactly one value.

```
//AND-state named C
conc state C {
  v0: lone A
  v1: one A
  ...

  init {
    no v0 // v0 should initially be an empty set
    one v1 // v1 should initially be a singleton set
  }
}
```

Figure 2.6: Example Initial Constraints in Dash

The meaning of a Dash model is a next relation over snapshots. A snapshot includes the set of current basic states called the **active** basic states, the set of transitions that have been taken in a big step, the set of events that have been triggered and the current values of variables. Dash has the notion of a **big step** that is composed of a series of small steps. Dash has a notion of a big step such that multiple transitions can be taken in response to environmental events. A **small step** in Dash consists of a transition being taken. Each concurrent region takes at most one small step within a big step. A Dash model will take a number of small steps until it is not possible to take any more small steps (or transitions) in a big step. Once a system reaches the end of a big step, environmental events can be generated and the system will enter a new big step. The environmental events persist throughout the big step.

The semantics of Dash specify that every concurrent state can take at most one transition in a big step. Once a transition has been taken in a big step, the system can take transitions that are **orthogonal** to the transitions that have been taken in a big. Two transitions are orthogonal if they are declared in different concurrent regions. That is, an AND-state cannot take a transition if it has already taken a transition in a big step or if any nested AND-states have taken a transition in a big step.

A semantic choice in Dash dictates that any non-environmental variable retains its value if it is not explicitly constrained in an action. This choice was made to mix the common semantics of declarative languages with the common semantics of control-oriented languages. In declarative languages such as Alloy, a variable that is not constrained in an action is allowed to change its value non-deterministically; a variable in control-oriented languages usually retains its values from the previous snapshot if it is unchanged in an action.

We will discuss the semantics of Dash embodied in their translation to Alloy in Chapter 4 when we discuss the translation of Dash+ to Alloy.

Farheen [19] and Serna [41] proposed significance axioms to find “large enough” instances of Alloy models of transition systems. The goal of significance axioms is to ensure that it explores interesting portions of the snapshot[1] space and avoids spurious counterexamples during the process of model checking. There are three significance axioms that have been used with Dash [41] models previously:

- **Reachability Axiom:** The reachability axiom ensures that every snapshot is reachable from the initial snapshot. As a result, an arbitrary snapshot that is not reachable by a transition originating from the initial snapshot will not be in the instance.
- **Operations Axioms:** The operations axiom ensures that every transition is represented at least once during model checking. That is, we want to ensure that the scope we use is large enough to cover every transition in a model.
- **Complete Big Step Axiom:** The complete big step axiom ensures all the big steps of the transition system must be complete. We want to analyze instances that end in a stable snapshot meaning that the instances we consider produce complete big steps.

Dash performs well-formedness checks to help modellers avoid mistakes when writing a Dash model. The well-formedness checks [3] in Dash are:

- Every top-level `state` must be declared as `conc`.
- If a model has a state hierarchy, then there must be one default child state defined
- Transitions cannot cross AND-state boundaries
- Either all children states at the same level of the hierarchy are concurrent or none at all

[1]The commonly used term is `state` as in ‘state space’, but we use the term `snapshot` instead to avoid confusion with the term `control state`.

- Only snapshot variable declarations can be primed
- Environmental events cannot be generated in a transition
- Environmental variables cannot be primed

2.3 Summary

Alloy is a language based on relational logic and set theory and consists of a set of relations with constraints. Dash is a language that extends Alloy with concepts from Statecharts for modeling transition systems with concurrent hierarchical states. The semantics of Dash give it a notion of big steps and small steps to define transitions that can be taken by reacting to changes in the system.

Chapter 3

Dash+ Syntax and Semantics

In this chapter, we introduce Dash+, which is an extension to Dash for modelling transition systems with replicated AND-states running concurrently. This chapter presents a description of the syntax and informal semantics of Dash+ for creating replicated AND-states and communication between replicated components using variables and buffers.

3.1 Replicated AND-States

Alloy lacks an explicit construct for describing transition systems. Dash extends Alloy to describe transition systems with hierarchical concurrent states but lacks a construct for creating replicated AND-states. Our goal with Dash+ is to create a construct for describing replicated AND-states with a means of communication between replicated components (similar to declaring a set of replicated processes in PlusCal or Promela). It should be noted that every process in a PlusCal or Promela model must be declared at the top level, but Dash+ provides the flexibility of describing systems with replicated components arbitrarily nested in a state hierarchy. In this section, we focus on the design decisions taken for creating replicated AND-states and the constructs for declaring them. We call the states in a replicated AND-state as **components** and we call other components in the replicated AND-state relative to one component as its **sister components**.

A replicated AND-state construct in Dash+ has the meaning of writing multiple identical copies of an AND-state in Dash with appropriate relabelling. Dash+ supports models with replicated and non-replicated AND-states running concurrently with each other. As a language design decision, we consider whether the replicated AND-states should be explicitly or implicitly parameterized.

Implicit parametrization means a replicated component would not have an explicit parameter. The modeller would specify the number of copies of a replicated AND-state in a command. Implicit parameterization has the advantage of avoiding the excessive syntax of needing to include that parameter for every locally declared dynamic variable. However, the use of implicit parameterization has its disadvantages. There would be no means for sister components to communicate directly with each other since we cannot refer to a specific sister component. The only means of communication would be through broadcasting a message using a global variable.

A better solution is to explicitly parameterize replicated AND-states but any dynamic variable within the component that is not parameterized refers to this (local) component's copy of the variable. We specify the signature that parameterizes an AND-state using square braces when declaring the respective AND-state. By using operations on this signature, we can access variables in sister components directly.

An example of the syntax for describing a Dash+ model with a replicated AND-state is shown in Figure 3.1. A replicated AND-state called `R` is declared with two sub-states (`A` and `B`). Declaring a replicated AND-state is similar to declaring an AND-state in Dash using the `conc state` keyword, but we append a parameter in square braces with a signature.

In Figure 3.1, the AND-state `R` is parameterized with the `Id` signature on line 4. Parameterizing the replicated AND-state `R` means that each copy of the replicated component will be associated with an element from the `Id` signature. We have taken inspiration from Promela for the Dash+ syntax in declaring replicated AND-states. In Promela, we use a parameter to specify the number of copies of a process to create; Dash+ specifies the set of elements that refers to replicated components. The number of replicated components is determined when the modeller sets the size of the `Id` signature for analysis. We will refer to the signature parameterizing a replicated AND-state as the **identifier** signature for a replicated component.

One more benefit of explicit parameterization is that we can easily model arrangements of components in linear orders, rings, etc. Since a replicated AND-state is parameterized by a set of elements, we can place Alloy constraints on this set and organize them as needed.

In Figure 3.1, we have used the ordering module of Alloy to specify a linear ordering of the `Id` set (line 1). Since the replicated AND-state `R` is parameterized by the `Id` set, the copies for this AND-state will be arranged in a linear order. We see a graphical illustration in Figure 3.2. In Figure 3.2, there are three elements in the identifier set: `Id0`, `Id1` and `Id2`. The `Id` set is ordered using the ordering module which creates a linear ordering

```
open util/ordering[Id]
sig Value {}
conc state R [Id] {
  // local variables
  v: lone Value
  p: one Id
  // local events
  event E {}
  // the default state for each copy
  default state A {
    trans T_A {
      do {
        // change the local var v
        one val: Value | v' = val
      }
      goto B
      send E
    }
  }
  state B {
    trans T_B {
      on E
      do {
        // change the local var v
        one val: Value | R[this]/v' = val
      }
    }
    trans T_C {
      do {
        // change the var v in copy identified by p
        one val: Value | R[p]/v' = val
      }
    }
  }
  init {
    no v
  }
}
```

Figure 3.1: An Example of a Dash+ model

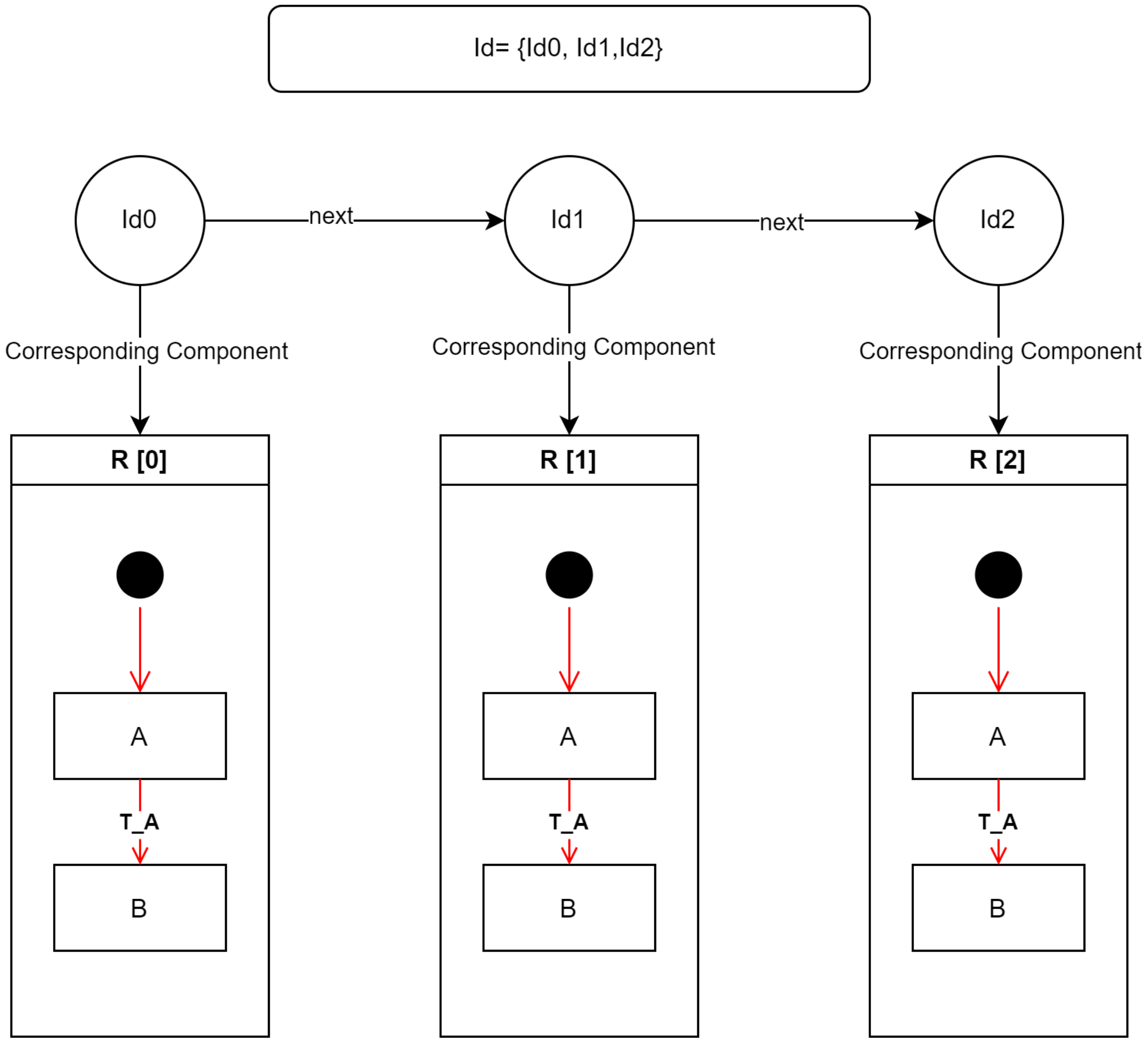

Figure 3.2: Graphical Illustration of Figure 3.1

for elements in the set. The first element in this set is `Id0` and the last element is `Id2`. Therefore, the copies of the replicated AND-state `R` are organized according to the ordering of the elements in the `Id` set. We may also specify for the replicated components to be arranged in a tree, ring, etc. structure by adding specific constraints on the identifier set for a replicated AND-state.

3.2 Dynamic Variables

We declare dynamic variables for a replicated AND-state in the same manner as we would for a non-replicated AND-state. Variables declared within a replicated AND-state are local to its copy of the replicated component. The meaning of each variable is a mapping from an element in the identifier signature to the value of the variable. The variables `v` and `p` in Figure 3.1 create mappings from each element in the `Id` signature to a value. Since a variable in a replicated AND-state is a mapping from an identifier element to a value, we can refer to a variable in a sister component by accessing the element that represents the sister component. As a result, we open the door for communication between replicated components, and between non-replicated and replicated components.

A replicated component can change the value of its local variable as seen in the transition `T_A` in Figure 3.1. In transition `T_A`, we change the value of the local variable `v` (line 15). Therefore, each copy of a replicated component for `R` will change their local variable `v` upon taking the transition `T_A`. We can also use the keyword `this` to refer to a replicated component taking a transition. Therefore, writing `v'` in an action is equivalent to writing `R[this]/v'` as shown in transition `T_B` (line 26).

One of our goals in Dash+ is to enable direct communication between sister components. Sister components in PlusCal and Promela can communicate with each other only by sending a message using global channels (buffers) and/or variables. In Dash+, a replicated component can communicate directly with a sister component without resorting to creating global variables for the sole purpose of communication. By using a signature as a parameter, replicated components can communicate directly with one or more sister component(s) and access their variables.

An example of a replicated component changing the value of a variable in a sister component is seen in transition `T_C` in Figure 3.1. In transition `T_C`, the replicated component taking this transition accesses the variable `v` in a sister component and changes its variable inside the action (`do`) statement (line 32). This sister component is the replicated AND-state that is identified by the singleton element in the local variable `p`.

We can use Figure 3.3 to better understand the concept of referring to a sister component to change the value of a variable in the sister component. The variables `v` and `p` are tuples with two elements. The first element is the identifier element and the second element is the variable value for the identifier element. Assume that the replicated component identified by the element `Id0` is taking the transition `T_C` when the relations have the values shown in Figure 3.3. The value of the local variable `p` is the singleton set `Id1` as seen in the tuple (`Id0`, `Id1`) in Figure 3.3 in `Id0`.

```
Id = {Id0, Id1, Id2}
Value = {Value0, Value1, Value2}
v = {(Id0, Value0), (Id1, Value1), (Id2, Value2)}
p = {(Id0, Id1), (Id1, Id0), (Id2, Id1)}
```

Figure 3.3: Mappings from an Identifier Element to Variables

The transition `T_C` changes the value of the variable `v` in the replicated component identified by the value of p in `Id0`, which is `Id1` to `Value2` with the resulting relation values shown below:

```
v = {(Id0, Value0), (Id1, Value2), (Id2, Value2)}
```

If we used a variable in place of `p` in transition `T_C` where the value of the variable consists of more than one element such as `Id1` and `Id2`, then our action statement in transition `T_C` would change the value of the variable `v` for all components associated with the identifier elements in the variable.

3.3 Events

An event in a replicated AND-state is declared without a parameter as in Dash. Events declared within a replicated AND-state are local to their respective copy as with variables. An environmental event of a replicated component can be triggered at the start of a big step, or a replicated component may trigger an internal event during a transition.

In Dash+, events can be used for communication between sister components and a replicated component can trigger an event in a sister component. We show the syntax for triggering an event in a sister component using a snippet of a Bit Counter model in Figure 3.4[1]. The Bit Counter model consists of a set of ordered replicated components. Each

[1]The full Bit Counter model is discussed in Chapter 6.

```
1  open util/ordering[Id]
2
3  conc state Bit [Id] {
4    event Tk1 {}
5
6    default state Bit1 {
7      ...
8    }
9
10   state Bit2 {
11     trans T_B {
12       on Tk1
13       goto Done
14       // next[this] returns parameter value
15       // for the next bit in the ordering
16       send Tk1[next[this]]
17     }
18     ...
19   }
20
21   state Done{}
22 }
```

Figure 3.4: A Parameterized Bit Counter

copy of the replicated AND-state `Bit` transitions from state `Bit1` to `Bit2` and triggers a transition in the next component in the order. We see an example of triggering an event in a sister component in transition `T_B` in the trigger (`send`) statement (line 16). The event to send is specified and a parameter is added using square braces where the sister component to trigger the event in is specified. In Figure 3.4, we trigger an event in the component that is next to the component (in the ordering module) taking transition `T_B`.

3.4 Buffers

The modelling languages PlusCal and Promela are useful for modelling distributed systems and protocols in which multiple actors communicate between themselves. A distributed system can consist of a set of clients and servers which communicate with each other.

Under normal circumstances, a distributed system will have more clients than servers [5]. A number of clients can send messages to a single server, and these messages will need to be queued such that the server can eventually respond to every request it receives. Therefore, we need a data structure that has the ability to queue and pop messages. PlusCal and Promela support sequences and buffered channels respectively that are declared either globally or locally within processes for the purpose of queuing messages. We wanted to provide Dash+ with constructs for defining a data structure that can store an ordered set of elements. We will refer to this data structure as a **buffer**.

A buffer in Dash+ is declared using the `buf` keyword with a parameter to specify the type of element that the buffer will store. We see an example of the process of declaring buffers in a Dash+ model in Figure 3.5. In Figure 3.5, we describe a buffer called `bufVar` that stores an ordered set of `Value` elements. We add an element to the back of the buffer `bufVar` (line 9) in the action statement in transition `T_A` and we remove an element from `bufVar` in the action statement in transition `T_B` (line 13). An upper bound for the size of a buffer is specified in a check or run command (line 18).

```
 1  sig Value {}
 2  pred predicate {}
 3
 4  conc state R [Param] {
 5    bufVar: buf[Value]
 6
 7    default state A {
 8      trans T_A {
 9        do one v: Value | bufVar.add[v]
10      }
11
12      trans T_B {
13        do bufVar.remove
14      }
15    }
16  }
17  // bufVar has a maximum size of 4
18  run predicate for 4 bufVar
```

Figure 3.5: Declaring Buffers in Dash+

3.5 Initial Constraints

The initial constraints in any state in a Dash+ model specify the initial values of variables. An `init` statement within a replicated AND-state can specify the initial values of variables in replicated components. In Figure 3.1, we specify that the local variable `v` should be initially an empty set (line 37). Therefore, every copy of the replicated AND-state `R` will have an empty set for the variable `v`.

In Dash+, we can declare an initial constraint that describes initial values for variables in specific components. For example, we may want every replicated component for `R` except the one identified by the first element in the ordered set `Id` to have a specific value in `v` . We can specify such a constraint in an initial constraint in the parent AND-state to the replicated component as seen in Figure 3.6. The initial constraint specified in Figure 3.6 states that every copy of the replicated component `R` except the one identified by the first element in the `Id` set should have an empty value in the variable `v` (line 11). We additionally state that the replicated copy identified by the first element in the `Id` set must have one or elements in the variable `v` (line 12).

```
1     open util/ordering [Id]
2
3     conc state C {
4       conc state R [Id]{
5         ...
6       }
7
8       init {
9         // first returns the first identifier element
10        // in the ordering
11        all ie : Id - first | no R[p]/ v
12        some R[first]/v
13      }
14    }
15
```

Figure 3.6: An Initial Constraint

3.6 Frame Problem

One semantic issue with a design decision is the frame problem for Dash+. As discussed in Chapter 2, a semantic choice in Dash dictates that any non-environmental variable retains its value if it is not explicitly constrained in an action.

In Dash+, a variable for a replicated component is a mapping from each copy of the replicated component to a value. Only one copy from a set of replicated components can take a transition in a step and may change the value of a local variable or a variable in a sister component. We want to stay consistent with the semantics of Dash such that a variable in a copy of a replicated component retains its value if it is not explicitly mentioned in an action.

There are limitations to the extent to which we can stay consistent with the semantics of Dash in having an unchanged variable retain its value. If a transition in Dash+ only changes a local variable in a copy of a replicated AND-state, we can specify that every sister component should retain the value of the variable that was changed.

```
trans T3 {
  do {
    one p: Id, val: Value | {
      R[p]/v' = val
    }
    one q: Id, val: Value | {
      R[q]/v' = val
    }
}
```

Figure 3.7: Changing a Variable in More than One Replicated Component

However, we can change the value of a variable in more than one copy of a replicated component in a transition as seen in Figure 3.7. There are two quantified expressions that each change the variable `v` in copies of a replicated component identified by the `Id` signature. One copy is identified by the bound variable `p`; the other copy is identified by the bound variable `q`. We want to ensure that every sister component retains its value for the variable `v` except as specified in the transition actions. The bound variables `p` and `q` are declared locally in their respective quantified expression meaning that we cannot access them from outside the quantified expression. In order to have unchanged variables retain their values in sister components, our semantics would need to specify the following:

1. Every copy of `R` except the copy identified by `p` retains its value in the variable `v`.
2. Every copy of `R` except the copy identified by `q` retains its value in the variable `v`.

which would result in an inconsistency if the copies of the replicated component identified by `p` and `q` are not the same.

We do not wish to choose semantics that may introduce an inconsistency in the model. Therefore, we choose a semantic in Dash+ in which unchanged variables in replicated components retain their values only when a transition changes a local variable in a replicated component. If there is a case in which a replicated component taking a transition changes a variable in a sister component, we will then issue a warning. The modeller will have to ensure that sister components retain the values in their variables.

3.7 Well-Formedness

Dash+ performs the same well-formedness checks as Dash. However, Dash+ also performs additional well-formedness checks due to the introduction of replicated AND-states. The additional well-formedness checks in Dash+ are as follows:

- The source/destination state for a transition in a copy of a replicated component cannot be a state in a sister component.
- The source/destination state for a transition in a copy of a replicated component cannot be a state in a non-replicated AND-state.
- The source/destination state for a transition in a non-replicated AND-state cannot be a state in a replicated AND-state.

3.8 Summary

Dash+ has constructs for creating transition systems with replicated components and non-replicated components that can be nested arbitrarily. Replicated and non-replicated components can communicate with each other by accessing variables or by triggering events. We introduce a syntax for declaring buffers in Dash+ to facilitate buffered communication between replicated components or between replicated and non-replicated components.

Chapter 4

Translation to Alloy

In this chapter, we discuss the steps required to translate a Dash+ model to Alloy, how parameterization is weaved into transitions, and the process of translating buffers in Dash+ to Alloy.

4.1 Background: Translating Dash to Alloy

This section will briefly focus on the translation of a Dash model to Alloy from Serna [41] and facilitate a discussion in the next sections of translating replicated components and buffers in Dash+ to Alloy. The translation to Alloy is decomposed into two steps:

- A translation of a Dash model to Core Dash
- A translation of a model in Core Dash to Alloy

4.1.1 Dash to Core Dash

A model in Dash is initially translated to Core Dash. Core Dash consists of the state hierarchy of a model, a set of transitions that have been completely elaborated with information, initial constraints, and state invariants. The translation to Core Dash is a valuable step because modellers in Dash can describe transitions using various shortcuts. For example, a modeller writing a Dash model can describe a transition without explicitly specifying a source (`from`) and destination state (`goto`). When a transition is described without a source (`from`) state or destination (`goto`) state, the translation to Core Dash will use the container state of the transition as the origin state and the destination state. Furthermore, the names of states, transitions, variables, and events are replaced with their fully qualified names. A **fully qualified name** is formed by following the state hierarchy of an element and separating state names with '/' and then appending the name of the element. Figure 4.1 shows a Dash model on the left with its Core Dash translation on the right. There are two transitions `T0` and `T1` that are declared without any origin or destination states. The translation to Core Dash in Figure 4.1 calculates the origin and destination states for the transitions and completes the fully qualified names for the states, transitions, and variables in the model.

```
sig Value {}
conc state C0 {
  v0: one Value
  v1: some Value
  event E0 {}

  default state S0 {
    trans T0 {}
  }

  state S1 {
    trans T1 {}
  }
}
```

```
sig Value {}
conc state C0 {
  C0/v0: one Value
  C0/v1: some Value
  event C0/E0 {}

  default state C0/S0 {}
  state C0/S1 {}

  trans C0/S0/T0 {
    from C0/S0
    goto C0/S0
  }
  trans C0/S1/T1 {
    from C0/S1
    goto C0/S1
  }
}
```

Figure 4.1: Translation from Dash to CoreDash

4.1.2 Core Dash to Alloy

A Dash model defines a next snapshot relation containing pairs. That is, the next relation is a set of tuples (with two elements) where the first element is a snapshot element that describes the current snapshot of the system; the second element in the tuple describes the snapshot of the system after taking a small step (or transition). A Dash model takes a sequence of small steps with cascading events until it reaches the end of a big step and is able to initiate another big step. We consider a Dash model to be stable at the start and end of a big step meaning that there are no more transitions that can be taken so new environmental events can be input.

A **snapshot** signature declaration describes relations that contain the information needed to describe the values of the system elements. That is, each atom in the snapshot signature maps to values at a point in the execution of a system. The translation of a CoreDash model to Alloy will result in the declaration of a snapshot signature with at least three relations:

- **conf**: the `conf` (or configurations) relation contains the set of active basic states
- **taken**: the `taken` relation contains the set of transitions taken in the big step so far
- **stable**: `stable` contains the value `True` if the snapshot is at the beginning/end of a big step

In addition to the relations stated above, the snapshot signature can have an **event** relation which is composed of a set of events that have been generated in a big step so far. We include the `event` relation only if events are declared within a Dash model. There are also relations in a snapshot signature for storing the values of variables declared within a Dash model. Figure 4.2 shows an example of a snapshot signature.

We create a sub-signature tree in Alloy to express state hierarchy. An abstract signature is used to represent a container state and a singleton signature refers to the basic control states in a container state. Figure 4.3 shows the representation of the state hierarchy in Figure 4.1 after the translation to Alloy. The container state `C0` is declared as an abstract signature (we use fully qualified names in the translation to Alloy), and the basic control states are declared as singleton signatures. In Figure 4.1, the basic state `S0` is translated to a singleton set called `C0_S0` (`one sig C0_S0 extends C0 {}`); the basic state `S1` is translated to a singleton set called `C0_S1` and so on. At any point during the execution of the model, the `conf` relation in the snapshot signature will map to a set of basic control states. From these basic states, we can determine the state that a concurrent region is in

```
sig Snapshot {
  // set of active basic states
  conf: set StateLabel
  // set of events that have been triggered so far
  events: set EventLabel
  // set of transitions taken in a big step so far
  taken: set TransitionLabel
  // True at the beginning/end of a big step
  stable: one Bool

  // Variables
  v0: one Value
  v1: some Value
}
```

Figure 4.2: Example Snapshot Signature

```
1  // State Hierarchy
2  abstract sig StateLabel {}
3  abstract sig C0 extends StateLabel {}
4  one sig C0_S0 extends C0 {}
5  one sig C0_S1 extends C0 {}
6
7  // Transitions
8  abstract sig TransitionLabel {}
9  one sig C0_S0_T0 extends TransitionLabel {}
10 one sig C0_S1_T1 extends TransitionLabel {}
11
12 //Events
13 abstract sig EventLabel {}
14 one sig C0_E0 {}
```

Figure 4.3: Signatures for Representing State Hierarchy, Transitions and Events in Alloy for Figure 4.1

at any level in the state hierarchy. The transitions and events that are declared within a model are represented as singleton signatures in Alloy (using fully qualified names) as seen in lines 8-14 in Figure 4.3.

Figure 4.4 shows the predicates that are created during the process of translating a Dash model to Alloy. The top-level predicate is the `small_step` predicate and it is a disjunction of predicates that we refer to as transition predicates. The `small_step` predicate relates two snapshots that satisfy a transition predicate within the `small_step` predicate. Every transition declared in a Core Dash model will be translated into one transition predicate.

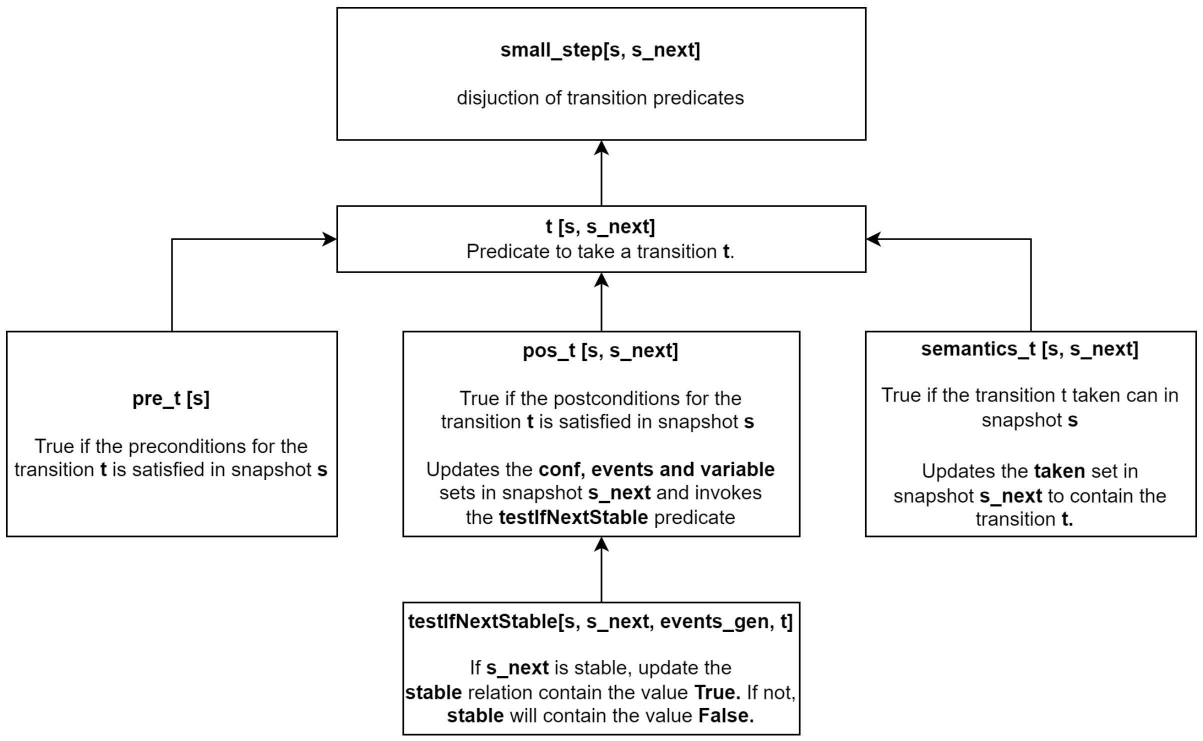
small_step[s, s_next]
disjuction of transition predicates
t [s, s_next]
Predicate to take a transition t.
pre_t [s]
True if the preconditions for the transition t is satisfied in snapshot s
pos_t [s, s_next]
True if the postconditions for the transition t is satisfied in snapshot s
Updates the conf, events and variable sets in snapshot s_next and invokes the testIfNextStable predicate
semantics_t [s, s_next]
True if the transition t taken can in snapshot s
Updates the taken set in snapshot s_next to contain the transition t.
testIfNextStable[s, s_next, events_gen, t]
If s_next is stable, update the stable relation contain the value True. If not, stable will contain the value False.

Figure 4.4: Overview of the Transition Predicates

A **transition** predicate is a conjunction of predicates. When all these predicates are satisfied, it means that the transition represented by this transition predicate is taken. These predicates are:

- **precondition:** The precondition predicate for a transition is satisfied if the source state is present in the `conf` (or configuration) relation of the source snapshot, any events needed to trigger the transition are in the `events` relation, and the current values of the variables satisfy the guard condition.
- **postcondition:** The postcondition predicate specifies constraints that update the `conf` (or configuration) relation of the destination snapshot with the destination state of the transition; the `events` relation is updated with any events that are triggered by the transition; the stable relation is updated to contain the value `True` if the system is stable after taking the current transition or `False` otherwise; the value of the variables are updated based on the Alloy constraints in the action statement. We evaluate the postcondition predicate relative to the current snapshot, `s` and the next snapshot, `s_next`.
- **semantics:** The semantics predicate enforces the Dash semantics of each concurrent region taking at most one transition within a big step. It places constraints on the `taken` relation and updates the `taken` relation based on the transition being taken. The semantics predicate is also responsible for ensuring that the transition taken has the highest priority amongst the enabled transitions.

One more important predicate that is declared during the translation to Alloy is the **testIfNextStable** predicate. The `testIfNextStable` predicate has constraints that are satisfied only if the system is stable after we take a transition. The postcondition predicate invokes the `testIfNextStable` predicate and updates the value of the stable relation to `True` if the `testIfNextStable` predicate is satisfied meaning that the system has completed a big step once the current transition is taken.

4.2 Translating Replicated AND-States

In this section, we discuss the major changes in translating replicated AND-states to CoreDash and from CoreDash to Alloy. In the next sections, we discuss how parameterization is weaved into the transitions, and the process of translating buffers (in 4.3).

A replicated AND-state in Dash+ means that we have multiple copies of an AND-state running concurrently. There were two approaches that we considered when determining the process of translating a replicated AND-state to Alloy. The simpler approach is to create multiple identical copies of the AND-state that we want to replicate (effectively translating from Dash+ to Dash with appropriate relabelling). Figure 4.5 shows an example of a CoreDash model (right) that is the result of creating identical copies of a replicated AND-state `R` (left). There are three identical copies of the replicated AND-state `R`: `R0`, `R1` and `R2` as specified by a modeller.

```
sig Value {}
conc state R [Id] {
  p: some Value
  q: one Id

  default state A {
    trans T {...}
  }
  state B {
    trans T {...}
  }
}
```

```
sig Value {}
conc state R0 {
  R0/p: some Value
  R0/q: one Id
  default state R0/A{}
  state R0/B {}
  trans  R0/A/T {...}
  trans R0/B/T {...}
}

conc state R1 {
  R1/p: some Value
  R1/q: one Id
  default state R1/A{}
  state R1/B {}
  trans R1/A/T {...}
  trans R1/B/T {...}
}

conc state R2 {
  ...
}
```

Figure 4.5: Creating Identical Copies of an AND-state

The number of copies to produce is based on the scope of the identifier set in a command. Creating multiple identical copies means that we do not need to weave parameterization into our translation. However, we cannot dynamically choose a sister component to communicate with based on the current snapshot. Any communication between sister replicated components would require us to broadcast a message globally (using global variables) that a sister component is able to access.

The alternate and better approach is to weave parameterization into our translation to Alloy which means that the variables and state hierarchy in a replicated component are parameterized. Figure 4.6 is a translation of a Dash+ model (left) to CoreDash+ (right) using the parameterization approach. In the translation to CoreDash+ (right) in Figure 4.6, the variables `p` (`R/p`) and `q` (`R/q`) map from an identifier element to their respective values.

```
sig Value {}
conc state R [Id] {
  p: some Value
  q: one Id

  default state A {
    trans T {...}
  }
  state B {
    trans T {...}
  }
}
```

```
sig Value {}
conc state R [Id] {
  R/p: Id -> some Value
  R/q: Id -> one Id

  default state R/A {}
  state R/B {}

  trans  R/A/T {...}
  trans R/B/T {...}
}
```

Figure 4.6: Translating to CoreDash+ with Parameterization

Figure 4.7 shows how the snapshot signature is translated for a replicated AND-state in `R`.

```
sig Snapshot {
  ...

  // Variables
  R_p: Id -> some Value
  R_q: Id -> one Identifier
}
```

Figure 4.7: Example Snapshot Signature for a Dash+ Model with a Replicated AND-state

The variables are sets of tuples with three elements. The first element in the tuple is the current snapshot of the system; the second element is an identifier element for the replicated component in which the variable was declared; the third element is the value of the variable for the respective identifier element.

Parameterizing variables allows the snapshot signature to store the value of every variable in each copy of a replicated component. The value of a variable in a replicated component can be accessed using a dot join operation on the variable. For example, the dot join operation `q.(s.p)` returns the value of the variable `p` for the replicated component identified by the value in the set `q` (the value of `q` in a replicated component is an identifier element) in the snapshot `s`. Therefore, a replicated component can communicate with a sister component by referring to the identifier element of the sister component and accessing the value in any of its variables.

In the following subsections, we define the concept of a level (4.2.1) and discuss the translation of replicated AND-states to Alloy (4.2.2).

4.2.1 Levels in a Dash+ Model

In Dash+, replicated and non-replicated AND-states can be arbitrarily nested. In translating a Dash+ model to Alloy, we need to consider the level at which every nested replicated and non-replicated AND-state is declared. We define the term **level** as:

1) For a replicated AND-state: The level of a replicated AND-state is the number of replicated AND-states above it in the state hierarchy plus one.

2) For a non-replicated AND-state: The level of a non-replicated AND-state is the number of replicated AND-states above it in the state hierarchy.

We say that a basic state is at a level$_i$ if the basic state is declared inside a replicated or non-replicated AND-state at level$_i$; we say that a transition is at a level$_i$ if the source state of the transition is declared inside a replicated or non-replicated AND-state at level$_i$; In a model with no replicated AND-states, every transition and basic state are at level$_0$.

In Figure 4.8, the non-replicated AND-state `C` is at a level of one as it is nested within one replicated AND-state `R`; the replicated AND-state `Q` is at a level of two as it is nested within the replicated AND-state `R`, and `Q` is itself a replicated AND-state; the replicated AND-state `S` is at a level of 3 since it is nested within `Q` (replicated AND-state) which is nested within `R` (nested replicated AND-state), and `S` is itself a replicated AND-state.

4.2.2 State Hierarchy

The representation in Alloy of the state hierarchy is used for four key parts of the meaning of the execution of a transition in Alloy. These are:

1) determining if the current snapshot contains the source state of a transition (as part of the precondition predicate)

2) determining how the `conf` (or configurations) relation changes by exiting the source state of a transition and entering its destination state (as part of the postcondition predicate)

3) determining which transitions are orthogonal to this transition (as part of the semantics predicate)

4) determining how the `taken` relation changes (as part of the semantics predicate)

In the Dash translation to Alloy, calculations are done within the translation process so that information about a transition can be represented statically within the equivalent Alloy model. The state hierarchy is represented through sub-signatures in Alloy with basic

```
//level 1
conc state R [IdR] {
  //level 1
  conc state C  {
    default state A {}
    ...
  }

  //level 2
  conc state Q [IdQ] {
    default state B {
      // level 3
      conc state S [IdS] {...}
      // level 2
      conc state P {...}
    }
  }
}
```

Figure 4.8: Level of Control States

states being declared as singleton sets that are subsets of abstract signatures that represent the container state for each basic state. Thus, item (1) is covered using a constraint of the form:

```
src_state in s.conf
```

within the precondition meaning that we check whether the source state of the transition is in the `conf` relation in snapshot `s` (`conf` is a mapping from a snapshot to a set of active states).

Item (2) is handled for the Dash translation by the process of calculating the basic states that are exited and the basic states that are entered by the transition and directly using these in the Alloy model with a constraint of the form:

```
s_next.conf = s.conf - {basic_states_exited} +
  {basic_states_entered}
```

within the postcondition. If a transition loops back to its basic source state, it will stay in its basic source state.

Item (3) is handled for Dash by calculating which transitions are orthogonal to transition `t` and a constraint is used in the Alloy model of the form:

```
no s.taken &
  (union of transitions within t's parent AND-state)
```

within the semantics predicate meaning that a transition cannot be taken if the AND-state state taking a transition has already taken a transition, or if a nested AND-state has taken a transition.

When the system is stable, item (4) is handled for Dash by updating the `taken` relation with a constraint of the form:

```
s_next.taken = t
```

within the semantics predicate meaning the `taken` relation (in the next snapshot) should only contain the transition `t`. We only need to store the transition `t` in the `taken` relation as we are starting a new big step since the system is stable when the transition `t` is being taken. When the system is not stable, item (4) is handled for Dash by updating the `taken` relation with a constraint of the form:

```
s_next.taken = s.taken + t
```

within the semantics predicate meaning that we add the transition `t` the `taken` relation (in the next snapshot, `s_next`).

Since we do not know the number of copies of each replicated process in Dash+, we have to weave parametrization through the above parts of the Alloy model for every replicated transition at any level in the state hierarchy. We considered two approaches to weaving parameterization: static and dynamic. In the static one, we try to mimic the Dash translation as closely as possible using static relations (ones not within the relations of the snapshot) to store the mapping from identifier elements to basic states. The static approach requires including copies of every basic state for each copy of a replicated AND-state. Therefore, the cardinality of the basic state signatures should match the number of copies of the replicated component in which the basic state was declared so that each identifier element can map to a unique copy of a basic state. This approach increases the number of atoms in the model and uses set cardinality which did not provide the level of performance that we achieved using the dynamic approach.

The dynamic approach to representing the state hierarchy for replicated components means retaining in the snapshot (the dynamic part of the model) information about the mapping from identifier elements to their current basic state. As in Dash, a basic state in a Dash+ model is translated to a singleton set while container states are represented as abstract signatures in Alloy (see Figure 4.3).

In order to retain information about the active basic state for AND-states and replicated AND-states that are arbitrarily nested, we require more than one `conf` relation. In the case of a replicated AND-state within a replicated AND-state, the configuration is a mapping from a parent identifier element to a child identifier element to a basic state. Therefore, we need to consider each possible arrangement for a Dash+ model with AND-states and replicated AND-states nested arbitrarily.

The number of conf relations in the snapshot depends on the number of different levels of the basic states (the level at which a basic state is declared) and each `conf` relation will have an arity equal to a level of the basic states. That is, we need `conf` relations to track the active basic state declared within each replicated or non-replicated component at every level. Therefore, the `conf` relations for a Dash+ model with nested replicated and non-replicated AND-states will appear as:

$\texttt{conf}_0$`: StateLabel`
$\texttt{conf}_1$`: Identifiers - > StateLabel`
$\texttt{conf}_2$`: Identifiers -> Identifiers - > StateLabel`
`...`

where $\texttt{conf}_0$ stores the active basic state for non-replicated AND-states at level$_0$; $\texttt{conf}_1$ stores the active basic state for non-replicated and replicated AND-states at level$_1$; $\texttt{conf}_2$ stores the active basic state for non-replicated and replicated AND-states at level$_2$. That is, we need a $\texttt{conf}_i$ relation for basic states declared inside an AND-state at level$_\text{i}$.

`Identifiers` is an abstract signature and every identifier signature is a subset of the `Identifiers` signature. Using the `Identifiers` signature as a parent signature means that the `conf` relation can store a mapping from identifier elements to their current active basic states for multiple replicated AND-states. Since `conf` stores a mapping from identifier elements to their current basic state, we can use singleton signatures to represent the basic states. An example of the set of tuples that conf$_1$ maps to for the dynamic approach can look like:

```
{(Snapshot1 , Id0 , A), (Snapshot1 , Id1 , B)}
```

meaning that the AND-state identified by the element `Id0` is currently in the basic state `A` and the AND-state identified by `Id1` is currently in the basic state `B`.

The Dash+ model in Figure 4.9 will require two `conf` relations:

```
conf₁: Identifiers -> StateLabel
conf₂: Identifiers - > Identifiers -> StateLabel
```

where $\texttt{conf}_1$ stores the active basic state for the non-replicated AND-state `C`; $\texttt{conf}_2$ stores the active state for each copy of the replicated AND-state `Q`. The $\texttt{conf}_2$ relation has an arity greater than that of conf_1 since the basic state `C` is at a level greater than the basic state `A`. Thus, the relation $\texttt{conf}_1$ stores the active state for each copy of the non-replicated AND-state `C` within `R`; the relation $\texttt{conf}_2$ stores the active state for each copy of the replicated AND-state `Q` within each copy of `R`.

```
//level 1
conc state R [IdR] {
  //level 1
  conc state C  {
    default state A {}
    state B {}
    trans T_A {...}
    trans T_B {...}
  }

  //level 2
  conc state Q [IdQ] {
    default state C {}
    state D {}
    trans T_C {...}
    trans T_D {...}
  }
}
```

Figure 4.9: Replicated Component within a Replicated Component

To better clarify how the `conf1` and `conf2` relations store the active basic states, Figure 4.10 shows an some example `conf1` and `conf2` tuples (and their meaning) for the Dash+ model in Figure 4.9. Figure 4.11 provides a visualization of the Dash+ model in Figure 4.9. The basic states in Figure 4.11 that are colored in green represent the active basic states as seen in the tuples in Figure 4.10.

```
IdR= {(IdR$_0$), (IdR$_1$)}
IdQ = {(IdQ$_0$), (IdQ$_1$)}
conf$_1$: {(IdR$_0$, A), // AND-state C nested within IdR$_0$ is in A
  (IdR$_0$, B)}
conf$_2$: {(IdR$_0$, IdQ$_0$, D),  // IdQ$_0$ nested within IdR$_0$ is in D
  (IdR$_0$, IdQ$_1$, D),
  (IdR$_1$, IdQ$_0$, C), // IdQ$_0$ nested within IdR$_1$ is in C
  (IdR$_1$, IdQ$_1$, D)}
```

Figure 4.10: Tuples for `conf` Relations

In Figure 4.10, `IdR`$_0$ and `IdR`$_1$ are identifier elements for the replicated AND-state `R`; `IdQ`$_0$ and `IdQ`$_1$ are identifier elements for the replicated AND-state `Q`.

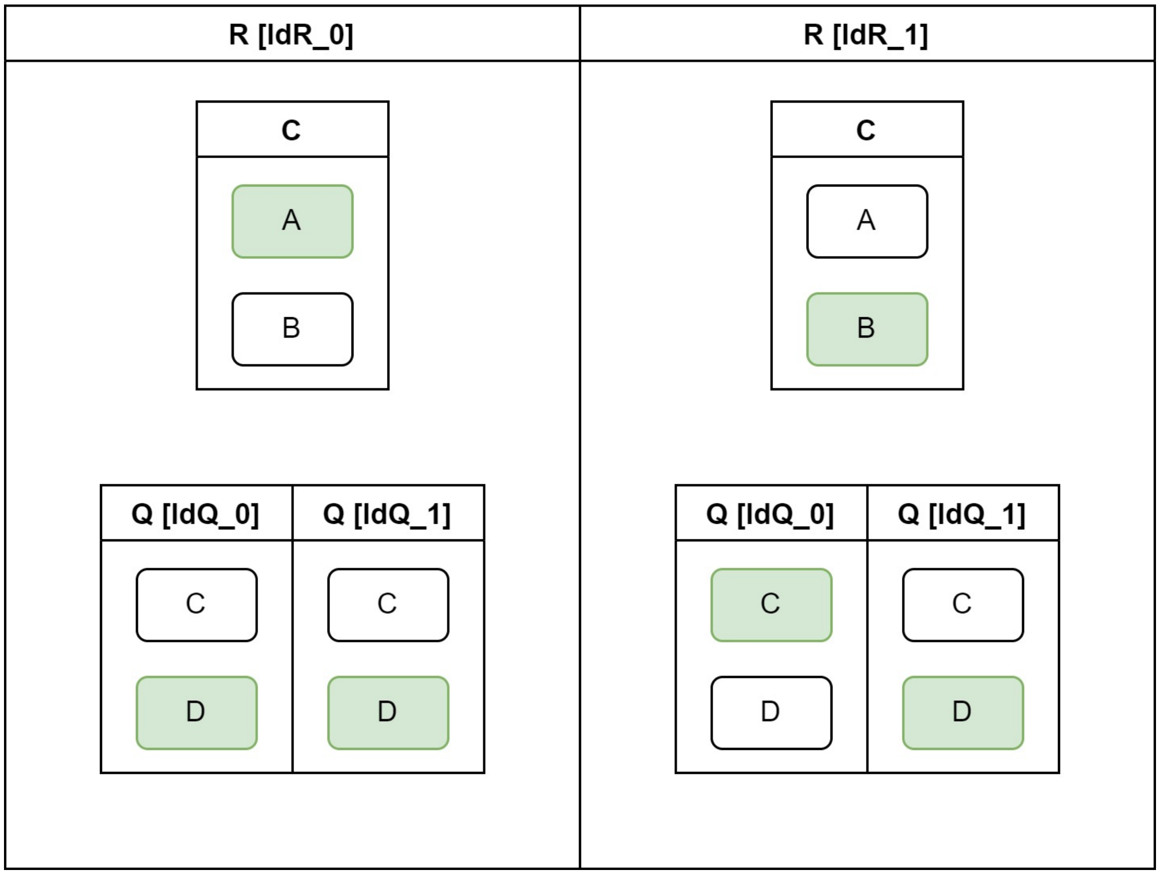

Figure 4.11: Visualization of the Dash+ model in Figure 4.9

We need to be able to handle item (1), item (2), item (3) and item (4) for every possible nesting of replicated and non-replicated components in a Dash+ model.

We handle item (1) (determining if the current snapshot contains the source state of a transition) for the dynamic approach using the constraint:

```
for a transition t at level$_i$, (src_state) in p$_1$.p$_2$...p$_i$.(conf$_i$)
```

where $\texttt{p}_i$ is a copy of a nested AND-state at level$_i$. The join operation $\texttt{p}_1$.$\texttt{p}_2$...$\texttt{p}_i$.($\texttt{conf}_i$) returns the current basic state of the replicated AND-state (or a non-replicated AND-state nested in a replicated AND-state) identified by the element $\texttt{p}_i$ nested at level$_i$. If the current basic state of the element in $\texttt{p}_i$ nested within the replicated AND-states ($\texttt{p}_1$, $\texttt{p}_2$, ..., $\texttt{p}_{\text{i-1}}$) is the source state of the transition, the precondition will be satisfied.

Item (2) (determining how the `conf` relation changes) will have the constraint:

```
for a transition t at level$_i$
  s_next.conf$_i$ = s.conf$_i$
    + (p$_1$ -> p$_2$ -> ... -> p$_i$ -> basic_state_entered)
    - (p$_1$ -> p$_2$ -> ... -> p$_i$ -> basic_state_exited)
```

where $\texttt{p}_i$ is a copy of an AND-state at level$_i$. We update the $\texttt{conf}_i$ relation with the basic destination state that $\texttt{p}_i$ has entered by taking transition `t` while removing the basic state that $\texttt{p}_i$ has exited.

Similar to the `conf` relations, we require more than one `taken` relation to store the transitions taken by each AND-state nested within replicated AND-states in a big step (the role of the `taken` relation is to store the transitions taken so far in a big step). The `taken` relations for a Dash+ model with nested replicated AND-states that each has basic states will appear as:

```
taken$_0$: TransitionLabel
taken$_1$: Identifiers -> TransitionLabel
taken$_2$: Identifiers -> Identifiers -> TransitionLabel
...
```

where $\texttt{taken}_0$ stores the transitions taken at level$_0$ in a big step; $\texttt{taken}_1$ stores transitions taken in a big step by each copy of an AND-state at level$_1$; $\texttt{taken}_2$ stores transitions taken in a big step by each copy of an AND-state at level$_2$ and so on.

As with Dash, each copy of a replicated AND-state can take at most one transition in a big step. A copy of a replicated AND-state can take a transition only if that copy has not yet taken a transition in a big step and if any nested replicated or non-replicated AND-states have not yet taken a transition in a big step. Alternatively, a non-replicated AND-state can take a transition only if it has not taken a transition in a big step and if any nested replicated or non-replicated AND-states have not taken a transition in a big step.

In Figure 4.12, it is permissible for a copy of `Q` to take the transition `T_B` only if it has not yet taken `T_B` in the big step and if any copy of the nested replicated AND-state `S` has not yet taken a transition in the big step. Similarly, the AND-state `C` can take the transition `T_C` only if `T_C` has not been taken in the big step and if the nested replicated AND-states `R`, `Q` and `S` do not have any copies that have taken a transition in the big step.

```
conc state C {
  default state A {
    conc state R [Id0] {
      default state B {}
      trans T_A {}
    }

    conc state Q [Id1] {
      default state C {
        conc state S [Id2] {...}
      }
      trans T_B {}
    }
  }

  state D {
    trans T_C {...}
  }
}
```

Figure 4.12: Replicated AND-state within an OR state

For a replicated AND-state identified by $\mathtt{p}_i$ at $\mathtt{level}_i$ taking a transition, item (3) (determining which transitions are orthogonal to this transition) will have the constraints for a transition declared within p_i:

```
no (pi <: p1.p2...pi-1.(takeni)) and
no (pi <: p1.p2...pi-1.(takeni+1)) and
no (pi <: p1.p2...pi-1.(takeni+2)) and
...
```

meaning that $\mathtt{p}_i$ cannot take a transition if it has already taken a transition or if a nested AND-state has taken a transition in a big step ($\mathtt{p}_1$, $\mathtt{p}_2$,...,$\mathtt{p}_{i-1}$ are identifier elements for any parent AND-states that $\mathtt{p}_i$ is nested within). We know that $\mathtt{p}_i$ has taken a transition if the domain of the set return by the dot join operation $\mathtt{p_1.p_2...p_{i-1}.(taken_i)}$ has the element $\mathtt{p}_i$. The replicated component identified by $\mathtt{p}_i$ also cannot take a transition if any nested replicated or non-replicated AND-state has taken a transition. If a nested AND-state at level_{i+1} has taken a transition, the domain of the dot join operation $\mathtt{p_1.p_2...p_{i-1}.(taken}$ $_{i+1}$) will contain the element $\mathtt{p}_i$ meaning that a nested AND-state within $\mathtt{p}_i$ has taken a transition.

To better clarify how orthogonality is decided, we use an example `taken1` and `taken2` tuples in Figure 4.13.

```
IdR=  {(IdR0)}
IdQ = {(IdQ0)} // identifier element nested within IdR0
IdS =  {(IdP0)} // identifier element nested within IdR0
taken1: {} // IdR0 has not taken a transition
taken2: {(IdR0, IdQ0, T_B), //IdQ0 nested in IdR0 has taken T_B
  (IdR0, IdS0, T_C)} //IdS0 nested in IdR0 has taken T_C
```

Figure 4.13: Example `taken` Sets

In Figure 4.13, the identifier signatures `IdQ` and `IdS` represent replicated AND-states (at $\mathtt{level}_2$) nested within the replicated AND-state identified by `IdR` (at $\mathtt{level}_1$). Therefore, the replicated AND-state identified by $\mathtt{IdR}_0$ can take a transition only if it has not yet taken a transition and if the nested replicated AND-states have not taken a transition so far in the big step. We know that $\mathtt{IdR}_0$ has not yet taken a transition since the $\mathtt{taken}_1$ set is empty ($\mathtt{IdR}_1$ it is at a $\mathtt{level}_1$). Since the domain of the tuples in the $\mathtt{taken}_2$ has the element $\mathtt{IdR}_0$ (meaning that a replicated AND-state nested within $\mathtt{IdR}_0$ has taken a transition), we can conclude that $\mathtt{IdR}_0$ cannot take a transition in the big step.

For a non-replicated AND-state `C` nested within a replicated AND-state identified by $\texttt{p}_i$ at level$_i$ taking a transition, item (3) (determining which transitions are orthogonal to this transition) will have the constraints:

```
no ((pᵢ <: p₁.p₂...pᵢ.(takenᵢ)) & (transᵢ)) and
  and no (pᵢ <: p₁.p₂...pᵢ₋₁.(takenᵢ₊₁)) and
  and no (pᵢ <: p₁.p₂...pᵢ₋₁.(takenᵢ₊₂)) and
  ...
```

where $\texttt{trans}_i$ is the union of the transitions in `C` (at $\texttt{level}_i$) and any transitions in nested non-replicated AND-states at $\texttt{level}_i$; the join operation $\texttt{p}_1\texttt{.p}_2\texttt{...p}_\texttt{i}\texttt{.(taken}_\texttt{i}\texttt{)}$ returns the set of transitions taken at $\texttt{level}_i$. Therefore, the constraints specify that `C` cannot take a transition if it (or a non-replicated nested AND-state at the same level) has already taken a transition or if any nested AND-states at lower levels have taken a transition in the big step.

We will now discuss the constraints needed for item (4) (determining how the `taken` set changes). If a copy of a nested replicated or non-replicated AND-state takes a transition while the system is stable, item (4) will have the constraint:

```
for a transition t at levelᵢ
  s_next.takenᵢ = (p₁ -> p₂ -> ... -> pᵢ -> t) and
  ∀ j in levels - i. no s_next.takenⱼ
```

where $\texttt{p}_i$ is a copy of an AND-state at level$_i$ taking a transition. If the system is stable, we specify that the $\texttt{taken}_i$ relation must only store the transition taken by $\texttt{p}_i$ nested at a level$_\text{i}$. Any other `taken` sets should be an empty set since the system is stable at the moment transition `t` is taken meaning that this is the first step of a big step.

If a copy of a nested replicated AND-state takes a transition while the system is not stable, item (4) will have the constraint:

```
for a transition t at levelᵢ
  s_next.takenᵢ = (p₁ -> p₂ -> ... -> pᵢ -> t) and
  ∀ j in levels - i. s_next.takenⱼ = s.takenⱼ
```

where $\texttt{p}_i$ is a copy of an AND-state at level$_i$ taking a transition. If the system is not stable, we update the $\texttt{taken}_i$ set with the transition taken by $\texttt{p}_i$ nested at a level$_\text{i}$. Any other `taken` sets should retain their values since the system is not yet stable and we need to keep a track of the transitions taken within the big step.

4.2.3 Weaving Parameterization into the Transition Predicates

We have seen the steps that are taken during the translation of Dash+ for parameterizing the state hierarchy and variables in a replicated component. The final step is to weave the parameterization into the transition predicates. We weave in the parameterization into the transition predicates by appending parameters such that a transition predicate is evaluated relative to the replicated component taking the transition. Since replicated components can be nested within replicated components, we may need to append more than one parameter to the transition predicates such that transitions are taken relative to a replicated component nested within other replicated components. Therefore, the transition predicate for the transition `t` within a nested replicated or non-replicated AND-state at level$_i$ would be described as:

```
pred t [s, s_next: Snapshot, p1: Id1, p2: Id2...pi: Idi] {...}
```

where $\texttt{p}_i$ is a copy of a nested replicated or non-replicated AND-state at level$_i$ taking a transition `t`.

We similarly append the parameters to the precondition, postcondition and semantics predicates such that the predicates are evaluated with respect to the replicated component taking the transition as shown below:

```
pred pre_t [s: Snapshot, p1: Id1, p2: Id2...pi: Idi]
  {...}
pred post_t [s, s_next: Snapshot, p1: Id1, p2: Id2...pi: Idi]
  {...}
pred semantics_t [s, s_next: Snapshot, p1: Id1, p2: Id2...pi: Idi]
  {...}
```

where $\texttt{p}_i$ is a copy of a nested replicated or non-replicated AND-state at level$_i$ taking a transition `t`.

The `small_step` predicate for a Dash+ model should be a disjunction over the transitions. Since the transition predicates have added parameters based on the level of nesting of AND-states, the `small_step` predicate can invoke a transition by passing in the identifier element of the AND-state taking a transition as an argument. To achieve a disjunction over all transitions of the model, we existentially quantify over the identifier sets as shown:

```
pred small_step [s, s_next: Snapshot] {
  some  $p_1$: $Id_1$, $p_2$: $Id_2$...$p_i$: $Id_i$ | {
    // for all t's at level 0
    ($\vee$ t[s, s_next]) or
    // for all t's at level 1
    ($\vee$ t[s, s_next, $p_1$]) or
    // for all t's at level 2
    ($\vee$ t[s, s_next, $p_1$, $p_2$]) or
    //...
  }
}
```

Figure 4.14 provides a visual illustration of the process of weaving in parameterization to the transition predicates for a Dash+ model with replicated AND-states.

While we have not shown it in the definition of the `small_step` predicate above, the `small_step` predicate can take a transition that retains the values of all the fields in the next snapshot. Since we want to create infinite traces of a Dash+ model, a Dash+ model takes a transition that retains the values of all the fields only if there are no transitions in the model that are enabled (meaning that the model has reached a point in which it cannot take any more transitions that were defined in the model).

We also use a fact that specifies two distinct snapshots cannot have fields with the same values. In Alloy, a snapshot is not a record but rather a distinct set of atoms. Because two snapshots with the same field values can be confusing to the user, we use a fact that specifies that two distinct snapshots cannot have fields with the same values.

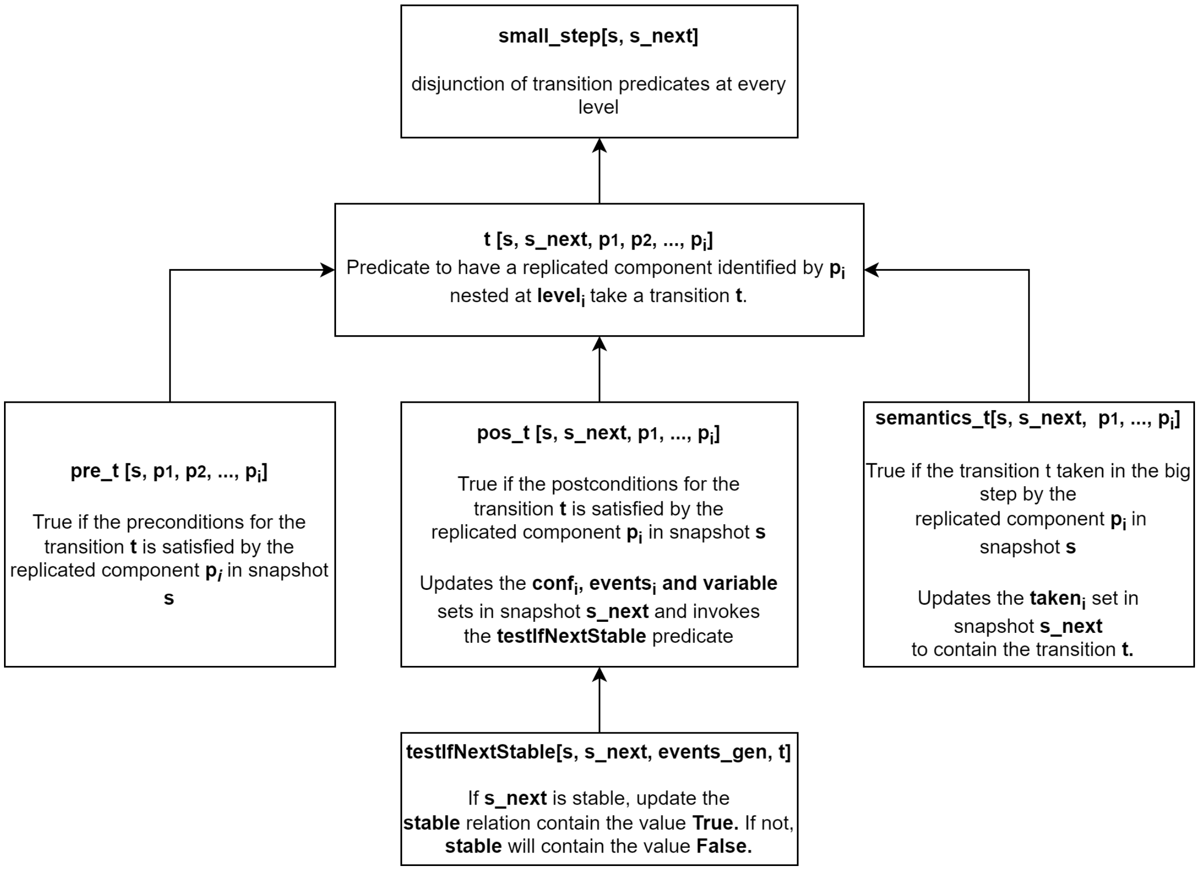
small_step[s, s_next]
disjunction of transition predicates at every level
t [s, s_next, p1, p2, ..., pi]
Predicate to have a replicated component identified by pi nested at leveli take a transition t.
pre_t [s, p1, p2, ..., pi]
True if the preconditions for the transition t is satisfied by the replicated component pi in snapshot s
pos_t [s, s_next, p1, ..., pi]
True if the postconditions for the transition t is satisfied by the replicated component pi in snapshot s
Updates the confi, eventsi and variable sets in snapshot s_next and invokes the testIfNextStable predicate
semantics_t[s, s_next, p1, ..., pi]
True if the transition t taken in the big step by the replicated component pi in snapshot s
Updates the takeni set in snapshot s_next to contain the transition t.
testIfNextStable[s, s_next, events_gen, t]
If s_next is stable, update the stable relation contain the value True. If not, stable will contain the value False.

Figure 4.14: Weaving in Parameterization

4.3 Buffers

This section will focus on the translation of buffers in a Dash+ model. The Dash+ model in Figure 4.15 has a replicated AND-state with two buffers: `bufA` and `bufB`. An element is added to `bufA` in the action statement in transition `T_A`; an element is removed from `bufB` in the transition `T_B`. A buffer variable is a relation in the snapshot signature with a mapping from an index signature to an element. We create an index signature for each buffer that is declared within a Dash+ model. The index signature is responsible for keeping a track of the location of each element in a buffer as each element in an index signature maps to a value in a buffer. Since buffers can be declared in any AND-state at any level of nesting, `buffer` variables in the snapshot signature will be declared as:

```
sig Snapshot {
  ...

  buf0: IndexA -> elem
  buf1: Identifiers -> IndexB -> elem
  buf2: Identifiers -> Identifiers -> IndexC -> elem
  ...
}
```

where $\texttt{buf}_0$ is a buffer declared at level_0 and using the $\texttt{Index}_A$ set for its indices; buf_1 is a buffer declared at level_1 and using the $\texttt{Index}_B$ set for its indices and so on.

The translation to Alloy in Figure 4.16 creates two index signatures: `BufIdxA` and `BufIdxB`. The elements in the `BufIdxA` are used as indices for `bufA`; the elements in `BufIdxB` are used as indices in `bufB`. The buffers `bufA` and `bufB` are included in the snapshot relation in Alloy. An example of the set of tuples for the snapshot relation for buffer `bufA` is:

```
{(Id0, BufIdxA0, Value0), (Id0, BufIdxA1, Value1),
  (Id1, BufIdxA0, Value1)}
```

where the tuple $(\texttt{Id}_0, \texttt{BufIdxA}_0, \texttt{Value}_0)$ means that the copy of `R` identified by $\texttt{Id}_0$ has the element $\texttt{Value}_0$ in the first index $(\texttt{BufIdxA}_0)$ in the buffer `bufA`; the tuple $(\texttt{Id}_0, \texttt{BufIdxA}_1, \texttt{Value}_1)$ means that the copy of `R` identified by $\texttt{Id}_0$ has the element $\texttt{Value}_1$ in the second index $(\texttt{BufIdxA}_1)$ in the buffer `bufA`.

We introduce a `buffer` module parameterized by both the index and value set of the buffer for performing operations on a `buffer` variable in Dash+. The `buffer` module defines predicates for performing operations on a `buffer` variable in Dash+. The predicates that perform operations on a `buffer` accept at least two arguments in the form:

```
conc state R [Id] {
  bufA: buf [Value]
  bufB: buf [Value]

  default state S {
    trans T_A {
      //add an element to buf0
      do one v: Value | bufA.add[v]
    }
    trans T_B {
      // remove an element from buf1
      do bufB.remove[v]
    }
  }
}
run pred for 4 bufA, 3 bufB
```

Figure 4.15: A Dash+ Model with Buffers

```
// Add the element e
add[R_cur: Index->elem, R_next: Index->elem, e: elem] {...}
// Remove an element
remove[R_cur: Index->elem, R_next: Index->elem] {...}
```

where R_{cur} is a `buffer` variable in the current snapshot and R_{next} is a buffer variable in the next snapshot. That is, the predicate places constraints on a buffer variable in the next snapshot based on the operation we want to perform in a buffer variable in the current snapshot.

To invoke a buffer predicate on a buffer $\texttt{buf}_i$ in a nested AND-state identified by $\texttt{p}_i$ at level_i, we use the following:

```
// Add the element value to buf_i
add[p_1.p_2...p_i.(s.buf_i), p_1.p_2...p_i.(s_next.buf_i)), value]
// Remove an element from buf_i
remove[p_1.p_2...p_i.(s.buf_i), p_1.p_2...p_i.(s_next.buf_i))]
```

where the join operation $\texttt{p}_1\texttt{.p}_2\texttt{...p}_i\texttt{.(s.buf}_i\texttt{)}$ returns the buffer variable $\texttt{buf}_i$ for $\texttt{p}_i$ in the current snapshot `s` which we pass to the predicate in the buffer module as an argument.

The scope of an index signature determines the maximum size of a buffer since we map

```
open util/buffer[Value,BufIdxA] as buffer0
open util/buffer[Value,BufIdxB] as buffer1

sig BufIdxA {}
sig BufIdxB {}

sig Snapshot {
...

  // buffer relations
  bufA: Id -> BufIdxA -> Value,
  bufB: Id -> BufIdxB -> Value,
}

pred pos_R_S_T_A[s,s_next: Snapshot,p0: Id] {
  // adding an element to bufA
  (one v: one Value |
    buffer0/add[p0. (s.bufA), p0. (s_next.bufA), v])
}

run predicate for 4 BufIdxA, 3 BufIdxB
```

Figure 4.16: Translation of Buffers

elements from the index signature to values. The modeler specifies a size for the buffers in a run or check command. In Figure 4.15, we specify that `bufA` should have a maximum size of 4 and `bufB` should have a maximum size of 3. During the translation to Alloy, we modify the command so that `bufA` is translated to `BufIdxA` (the index set for `bufA`) and `bufB` is translated to `BufIdxB` as shown in Figure 4.16.

We introduced the `buffer` module as it holds a few advantages over Alloy's built-in `sequence` module. There is only one index set in the `sequence` module meaning that every buffer containing the same set of values must be of the same size. The buffer module alleviates this issue by using a new index set per buffer. Furthermore, the sequence module does not have constructs for removing an element or adding an element to the front of a buffer. The `buffer` module adds constructs for removing an element, and adding an element to the front of a `buffer`.

4.4 Summary

In translating Dash+ to Alloy, we consider all the possible ways in which replicated and non-replicated AND-states can be nested and how the translation to Alloy handles each possible nesting of replicated and non-replicated AND-states. The translation process also handles the translation of buffers declared at any level in the hierarchy and connects each `buffer` to the buffer module for performing the operations on buffers.

We have implemented our translation in the Alloy Analyzer (version 6)[1]. A user has menu options to create a Dash+ model and translate it to Alloy or directly run a Dash+ model (which first translates a Dash+ model to Alloy and then runs it). Instances of the Dash+ model are presented in the regular Alloy visualizer.

[1]The implementation of Dash+ into the Alloy Analyzer (version 6) is available at https://github.com/WatForm/org.alloytools.alloy/tree/dashplus

Chapter 5

Model Checking in Dash+

Model checking is a method for checking temporal properties of a transition system. This chapter will focus on the different methods that are available in Alloy for model checking and how we make Dash+ compatible with different model checking methods. The semantics of Dash creates a transition system defined by an **init** predicate and a **small_step** predicate, which do not introduce any relations themselves. In each section of this chapter, we discuss how to link these predicates to the creation of the next state relations for three different model checking methods in Alloy.

In model checking, we check for safety and liveness properties [2] of a trace. A safety property specifies that "something bad will not happen" and a liveness property specifies "that something good must eventually happen".

As in Alloy, all Dash+ models have bounds for every signature in the model resulting in the creation of a finite state space. Bounded model checking considers a finite prefix of a path where the length of the path is bounded by an integer [6]. As a result, bounded model checking (BMC) checks the part of the finite state space that is reachable by a path of a fixed length. Unbounded model checks the entire finite state space as permitted by available memory. In this thesis, we use methods for bounded model checking in Alloy (of paths with loops) to check for properties of Dash+ models.

5.1 Model Checking using the Traces method in Alloy

The traces-based method of bounded model checking is the classical method of performing BMC in Alloy [26]. Typically, we constrain the state signature in an Alloy model using Alloy's built-in ordering module to perform bounded model checking in Alloy. The ordering module defines two private variables: `Next` and `First`. The variable `Next` relates two elements in the order and the variable `First` maps to the first element in the order. There are two functions `next` and `first` that act as an interface to the private variables and we use these functions to connect the semantics of Dash to the ordering module.

A consequence of using Alloy's ordering module is that the traces cannot contain loops. The snapshots we use to represent Dash semantics in Alloy are not a record structure but are a set of elements with relations mapping to atoms which represents the state of the system. The Dash semantics include constraints that specify that two snapshots cannot map to the same set of values. As a result, the traces that are considered by the ordering module must have only distinct snapshots meaning that we cannot have loops in a trace. Since we are unable to loop to a previous snapshot, the use of the ordering module for bounded model checking will only give us finite paths.

One solution to the issue of having only finite paths is to use the built-in `traces` module[16]. The traces module extends the ordering module, but the next relation includes a loop such that the last state can loop to the previous snapshot. This modification allows BMC to consider traces of an infinite length. While the traces considered using the traces module have an infinite path, the number of snapshots in the trace is bounded by an integer. Therefore, the finite state space that we explore using the traces module is the same as we would explore using the ordering module. The only difference in using the traces module is that we loop back to a snapshot that we have previously visited.

Figure 5.1 shows how the `next` and `first` functions are used to connect a Dash model to the traces module which defines the same variables and functions as the ordering module. We use constraints to specify that the `First` relation (mapping to the first snapshot in the trace) should satisfy the `init` (initial) predicate; the `Next` relation must satisfy the `small_step` predicate meaning that system transitions from one snapshot to the next snapshot in the snapshot trace. Therefore, we avoid adding any extra relations to the model. Constraining the snapshot signature using the traces module means that a total ordering is imposed on the snapshot signature by forcing pairs of consecutive snapshots to be related using the relation `Next`.

Once we have connected a Dash+ model to the traces module, a trace can be created for a model. We can check for safety or liveness properties in all traces of a model using Alloy

```
open util/traces [Snapshot] as snapshot

fact {
  init[snapshot/first] and
  (all s: one Snapshot - snapshot/last |
    small_step[s, s. (snapshot/next)] )
}
```

Figure 5.1: Connecting a Dash+ model to the Ordering Module

expressions to describe the properties. Figure 5.2 shows an Alloy model with a liveness (line 9) and safety property (line 14). The liveness property specifies that the variable `v0` should eventually have one or more values in some snapshot of the trace; the safety property specifies that the variable `v1` should always map to exactly one value throughout the trace in all snapshots of the trace.

```
 1 sig Snapshot {
 2   v0: set Int
 3   v1: set Int
 4   ...
 5 }
 6 ...
 7 // Liveness Property
 8 assert liveness {
 9   some s: Snapshot | (some s.v0)
10 }
11
12 // Safety Property
13 assert safety {
14   all s: Snapshot | (one s.v1)
15 }
```

Figure 5.2: Traces Properties

5.2 Transitive Closure Based Model Checking (TCMC)

Transitive-closure-based model checking (TCMC) is a symbolic representation of the semantics of computational tree logic with fairness constraints (CTLFC) in Alloy [20]. TCMC is defined over transition systems. A counterexample to a property in TCMC returns a transition system that satisfies the model but fails to satisfy an assertion as a counterexample.

Kember et al. [31] extends TCMC to Path TCMC which requires that the counterexample returned is a path of the model that does not satisfy the property. Using Path TCMC makes it easier to understand a counterexample since we can directly extract the path from a transition system with branching paths in which an assertion was violated. We use the Path TCMC module for model checking using TCMC. The Path TCMC module defines two private variables: `S0` and `sigma`. The variable `S0` maps to a set of initial states and the variable `sigma` relates snapshots to each other. Similar to the ordering module, there are two functions that are used to connect a model to the Path TCMC module: `ksS0` and `ksSigma`. The function `ksS0` acts as interface to the `S0` variable and `ksSigma` acts as an interface to the `sigma` variable.

Figure 5.3 shows the constraint that connects the semantics of Dash+ to the Path TCMC module. We specify that `ksS0` maps to the set of snapshots that satisfy an initial constraint; `ksSigma` is the transition relation and is satisfied by the `small_step` predicate meaning that two snapshots that are related by `ksSigma` must satisfy the `small_step` predicate. Therefore, we avoid adding any extra relations to the model by using the semantics of Dash+ to create a transition system.

```
fact tcmc {
  //ksS0 satisfies init (initial) constraints
  (all s: one Snapshot | { s in ctl/ks_s0 } <=> init[s])
  //ksSigma satisfies small_step predicate
  (all s,s_next: one Snapshot |
    ({ s -> s_next } in ctl/ks_sigma)  <=>
    small_step[s, s_next])
}
```

Figure 5.3: Connecting a Dash+ model to the CTL module

Properties are defined using the temporal operators and path quantifiers of CTLFC. Each CTLFC operator is a function that defines a set of snapshots that satisfies the formula for the CTLFC formula. Atomic propositions are written using set comprehension for a set of snapshots. Figure 5.4 shows an Alloy model with a liveness (line 10) and safety property (line 17). The liveness property specifies that the variable `v0` should eventually have one or more values in all traces; the safety property specifies that the variable `v1` should always map to exactly one value throughout all traces.

When model checking, the number of snapshots in the command is the number of snapshots in the transition system and not the length of the paths explored. Since the transition system include branching paths, the branching paths can be quite short. TCMC may consider a finite prefix of a path that does not satisfy a liveness property by itself but it is possible that all extensions to the prefix satisfy the liveness property. For example, the liveness property in Figure 5.4 (line 10) may return a transition system as a counterexample in which the branching paths do not have the length to explore interesting behaviors of the model and satisfy the liveness property. As a result, we have made the decision to not check for liveness properties in our case studies in Chapter 6 when using TCMC for model checking.

```
1  sig Snapshot {
2    v0: set Int
3    v1: set Int
4    ...
5  }
6  ...
7  // Liveness Property
8  assert liveness {
9    // v0 has on all paths, eventually one or more value
10   af[{s: Snapshot | (some s.v0)}]
11 }
12
13 // Safety Property
14 assert safety {
15   // v1 has on all paths, in every snapshot,
16   // exactly one value
17   ag[{s: Snapshot | (one s.v1)}]
18 }
```

Figure 5.4: TCMC Properties

5.3 Electrum

Electrum [7] is an extension of Alloy that adds Linear Temporal Logic (LTL) operators to the relational logic in Alloy and has been implemented into the Alloy Analyzer (version 6). The Alloy Analyzer supports bounded and unbounded model checking of Electrum specifications where bounded model checking is performed by encoding into SAT via kodkod and unbounded model checking is performed by encoding into NuSMV [12]. Unbounded model checking with encoding into NuSMV checks for traces with infinite length but with finite state spaces. Electrum uses Linear Temporal Logic connectives such as `eventually` and `always` to express properties of traces for a model.

In an Electrum model, signatures and fields declared using the `var` keyword means that they are dynamic and their valuation may change over time. Within a fact, a signature or field that is declared using the `var` keyword can be primed to specify its value in the next step. Furthermore, Electrum specifications are interpreted over infinite length paths meaning that an Electrum specification should not have a state with no outgoing transition. An Electrum specification that is not able to loop back to a previous state will be inconsistent.

We have added support to convert a Dash+ model to an Electrum specification. A Dash+ model that has been converted to Electrum does not use the snapshot signature because dynamic variables can be declared in Electrum. Instead, the `conf`, `taken`, `events`, buffers and variable relations (declared in the snapshot signature for non-Electrum models) for non-replicated components at $\texttt{level}_0$ are translated to signatures declared using the `var` keyword meaning that they change their value over time. The `conf`, `taken`, `events` relations for AND-states not at $\texttt{level}_0$ are declared as `var` fields in the `Identifiers` signatures (the parent signature for all identifiers of replicated processes). Variables and buffers that are declared within replicated components are declared as `var` fields in their respective Identifier signature.

Figure 5.5 compares the identifier signature and snapshot signatures in Dash+ models that have been translated to Alloy and Electrum. The Dash+ model that has been translated in Figure 5.5 has one replicated AND-state `R` parameterized by the `Id` signature with two variables `v0` and `v1`. In the translation to Electrum, the `conf1` and `taken1` relations are declared inside the `Identifiers` signature as a `var` relation. The variables `R_v0` and `R_v1` are declared as `var` relations within the `Id` signature as these variables are declared inside a replicated AND-state parameterized by `Id`.

When used in an action, each of the dynamic vars is primed when we want to refer to their value in the next snapshot. The `conf` and variable relations are primed in the

```
// Parent Signature for all
// Identifier Signatures
sig Identifiers extends univ {}
sig Id extends Identifiers {}
sig Snapshot extends univ {
  conf1: Identifiers -> StateLabel,
  taken1: Identifiers -> TransitionLabel,
  stable: one boolean/Bool,
  R_v0:Id ->one Int,
  R_v1: Id ->some Int
}
```

```
// Parent Signature for all
// Identifier Signatures
sig Identifiers extends univ {
  var conf1: set StateLabel,
  var taken1: set TransitionLabel
}
var sig stable in Bool {}
sig Id extends Identifiers {
  var R_v0: one Int,
  var R_v1: some Int
}
```

Figure 5.5: Snippet of a Dash+ model translated to Alloy (left) and Electrum (right)

postcondition predicates and the `taken` relations are primed in the semantics predicates such that Dash+ models converted to Electrum can change their value over time.

As shown in Figure 5.6, the `init` and `small_step` predicates are invoked without any parameters as a Dash+ model translated to Electrum does not use the snapshot signature. Figure 5.6 shows the constraints needed to create traces for an Electrum model. We specify an initial state using the initial conditions of the model and that the small step predicate should be true between all the states in the trace.

```
fact traces {
  init and
  always small_step
 }
```

Figure 5.6: Creating a Trace in Electrum

Properties for Dash+ models specified in Electrum are written using Linear Temporal Logic connectives. We can specify whether a property holds eventually in a trace (liveness) or will always have to hold throughout the trace (safety). Figure 5.7 shows an Electrum model with a liveness (line 7) and a safety property (line 12). The liveness property specifies that the `var` sig `v0` should eventually have one or more values in some state in all traces;

```
var sig v0 in Int {}
var sig v1 in Int {}

...
// Liveness Property
assert liveness {
  eventually (some v0)
}

// Safety Property
assert safety {
  always (one v1)
}
```

Figure 5.7: Electrum Properties

the safety property specifies that the `var` sig `v1` should always map to exactly one value throughout all traces.

5.4 Summary

Dash+ offers three model checking options: traces-based model checking, transition-closure-based model checking and Electrum. The traces-based model checking option defines a path of an infinite length (with loops) but with a limited scope on the number of snapshots; the TCMC model checking method defines a transition system with a limited number of snapshots; Electrum interprets specifications over an infinite sequence of states and considers paths of an infinite length (by having loops in the trace). For model checking using the traces-based or TCMC model-checking methods, we connect our model to the traces or path TCMC module using functions and mitigating the need to add any extra relations to the model. Alternately, a Dash+ model can be translated into an Electrum specification for model checking using Electrum keywords.

Chapter 6

Case Studies

This chapter presents the case studies that we have developed to demonstrate the features of Dash+. We use the model checking options in Dash+ to check for properties of each of the models in this chapter and point out the Dash+ features that each model exploits.

For each of the case studies that we present, we use significance axioms (described on pg.12) to choose a scope for checking properties to ensure that we explore interesting behaviors of the model. The operations axiom is used to select a scope that is large enough for every transition in a model to be taken; the complete big step axiom is used to ensure that instances include complete big steps.

We analyzed the case studies on a computer with an Intel® Xeon® Processor E3 v5 processor running at 3.5 GHz. Each of our case studies can be found here.

6.1 Chord

Chord is a distributed hash table that was introduced in [44] and distinguished itself from other peer-to-peer networks using its provable correctness and performance. A Chord network consists of nodes that form a ring structure. Nodes that are a part of the ring structure are called **member** nodes. Member nodes are referred to as **live** nodes and non-member nodes as **dead** nodes. A dead node may become a member by joining the ring structure through a **join** operation. Every node has a unique identifier with member nodes having successor and predecessor pointers. Figure 6.1 is an illustration of a Chord network with five member nodes that form an ordered ring structure through the successor and predecessor pointers. A node that is colored green is a live node (it is a member of

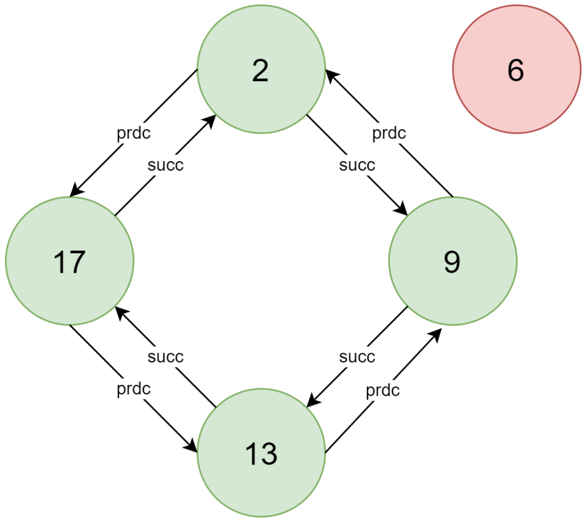

Figure 6.1: An Ideal Chord Ring Structure [48]

the ring); a node that is colored red is a dead node. The ring structure in Figure 6.1 is in an ideal state meaning that there is only one ring and the member nodes in the ring are ordered.

In Chord, a member node can fail at any point and cause disruptions in the ring structure. As a result, Chord has a ring-maintenance protocol that is responsible for correcting gaps in the ring structure and returning it to an ideal state. It is claimed in [44] that the ring-maintenance protocol is provably correct meaning that it will eventually fix any disruptions in the ring structure. However, Zave [48] has shown that the ring-maintenance protocol is not correct. Thus, Zave modelled a correct version of the ring maintenance protocol of Chord in Promela and Alloy and its correctness has been proven using invariants in the Alloy Analyzer and Spin Model Checker. The correct version of Chord modelled in Alloy consists of the following operations:

$$stabilize, notify, reconcile, update, flush, join$$

that are responsible for repairing gaps in the ring. These operations are atomic and change the state of a node. As shown in Figure 6.2, Zave's version of Chord uses a `NetState` signature that is used to define the global state of the system and a `time` variable is used to model the passage of time for the system.

```
sig NetState {
  time: Time,
  members: set Node,
  succ: members -> one Succs,
  prdc: members -> one Node,
  status: members -> lone Status,
  saved: members -> lone Node,
  bestSucc: members -> lone members,
  principals: set members
  status.Status = saved.Node
  ....
}

pred StabilizeFromSuccessor [s, s': NetState, t: Node] {
  -- PRECONDITIONS
  t in s.members
  no t.(s.status)
  -- POSTCONDITIONS
  s'.time = next [s.time]
  s'.members = s.members
  s'.prdc = s.prdc
  let succ1 = t.(s.succ).list[0] |
  succ1 ! in s.members =>
  (  (some u: Succs |
  s'.succ = s.succ ++ (t -> u)
  ...
  && s'.status = s.status ++ (t -> Stabilizing)
  && s'.saved = s.saved ++ (t -> succ1.(s.prdc))
  ...
}

```

Figure 6.2: Correct Version of Chord as Modelled in Alloy by Zave [48]

```
open util/ring[Node]
conc state System {
  members: Node
  conc state N [Node] {
    frst: lone Node
    scnd: lone Node
    prdc: lone Node
    ..
    default state Live {
      trans StabilizeFromSucc {
        when {
          no status
        }
        do {
          // The Successor is dead
          frst !in members implies {
            frst' = scnd
            scnd' = nextNode[scnd]
            ...
          // The successor is a member
          } else {
            frst' = frst
            scnd' = N[frst]/frst
            // The successor's predecessor is better
            (some N[frst]/prdc and ...) => {
              status' = Stabilizing
              ...
            }
          }
        }
      }
      ....
    }

  }
  ...
}
```

Figure 6.3: Chord in Dash+

We have modelled the correct version of the Chord protocol in Dash+[1] and a snippet is shown in Figure 6.3. In the Dash+ model of Chord, we define a state hierarchy such that member nodes are in a live state and non-member nodes are in a failed state. The Chord operations are defined as transitions that a live node can take based on the status of each node and the global state of the system. A node fails when it takes a fail transition and transitions to the failed state. Failed nodes may join the ring structure through a transition (to the live state) defined in the failed state. In the actual implementation of Chord, nodes periodically check the status of their successor and predecessor nodes and update the structure of the ring based on the state of the successor and predecessor nodes. In Dash+, we imitate this behavior by having each node take a transition in a big step and perform an operation to resolve any issues in the ring structure by checking the status of sister nodes.

The property that we want to check of Chord in Dash+ is that the ring structure always becomes ideal (there is only one ring and the member nodes in the ring are ordered). The ideal property uses the predicates defined in Figure 6.4 to specify that the system is in an ideal state. Figure 6.4 shows the predicates that we use to check if a Chord system has reached an ideal state (using the traces-based model checking method).

```
// Returns the members that form a ring
fun ring [s: Snapshot] : some Node {
  {m: s.System_members | m in m.^(succ[s])}
}

// Members form at most one ring
pred atMostOneRing [s: Snapshot] {
  (all m1, m2: ring[s] | m1 in m2.^(succ[s]))
}

// Members are ordered
pred orderedRing[s: Snapshot] {
  all disj m1, m2, mb: ring[s] |
    m2 = m1.^(succ[s]) implies not between[m1, mb, m2]
}
```

Figure 6.4: Properties of Chord in Alloy

[1]The complete model can be found in Appendix A.

Chord				
Property	Snapshot Scope	Node Scope	Traces (s)	Electrum (s)
Ideal	15	4	120	44
Ideal	15	5	1361	147

Table 6.1: Time Taken to Check Chord Properties (s = seconds)

The time taken to analyze the properties for Chord is shown in Table 6.1. For model checking of the Chord model in Dash+, Electrum performs better than the Traces method. We also found it easier to write the property that we need to check the Chord model in Electrum since we can use the LTL operators of Electrum to specify that the system should eventually reach an ideal state if there are no more join or fail events. We do not use TCMC since the traces do not have loops and we may not have a trace in which Chord reaches an ideal state.

The Dash+ features that are highlighted using this case study are:

1) *Locality of description*: Dash+ gives us the opportunity to model Chord with respect to a single node. We specify how a single node takes a transition based on the state of its successor and predecessor nodes. In Zave's model of Chord, each dynamic variable of a Node is modelled as a function from the global state and node identifier to the value. This extra parametrization is tedious and makes it difficult to understand the model from the point of view of one node. Chord has also been modelled in Electrum [8] and similarly describes a global state and global variables to specify how the ring structure changes over time.

2) *Communication between sister components*: Each node in Chord directly communicates with successor and predecessor nodes by accessing the values in their variables and checking if the successor or predecessor pointer needs to be updated.

3) *Arranging replicated components in a ring*: We arrange the nodes in a ring structure by constraining the `Node` signature using the ring module[2]. Additionally, the `frst`, `scnd` and `prdc` variables in the replicated AND-state `Node` are defined as mappings from a node to a node. Since the `frst`, `scnd`, and `prdc` variables store a node-node mapping for every node, we will have arranged the nodes in a ring structure once the Chord system is ideal.

4) *Use of named control states*: The basic state of a node in the Dash+ model for Chord determines whether it is a live node or a failed node. In Zave's version of Chord,

[2]The ring module is an extension to the ordering module in which the last element in the order points to the first element in the order.

we have to check whether a node in a member set to determine whether the node is a live or failed node.

6.2 Distributed Spanning Tree Algorithm

The Distributed Spanning Tree Algorithm [42] consists of a set of nodes that form a network topology starting from a distinguished root node. Each node has a level and is connected to a neighboring node that is identified using a parent pointer. The presence of a parent pointer connecting two neighboring nodes creates a network topology taking the form of a connected undirected graph [37].

The algorithm that creates the connected network topology starts at the root node which is selected prior to the algorithm being run. The root node assigns itself a level of zero and sends a message to another node with its identifier and level. A node that receives a message assigns its parent pointer to the node sending the message and assigns its level to one plus the level of the node that sent the message. The process of sending messages to nodes is repeated until every node has assigned itself a level and a parent

We have modelled the Distributed Spanning Tree Algorithm in Dash+[3] using a replicated AND state `N` parameterized by the set `Node` where each copy of `N` represents an individual node. The replicated AND-state `N` has a `parent` variable that maps to a node, and a `level` variable mapping to an element from a `Level` set which is constrained by Alloy's ordering module. The `Level` set is constrained using the ordering module such that we can describe a specific level for each node. Figure 6.5 shows a snippet of the Distributed Spanning Tree algorithm in Dash+. A global variable `root` identifies the root node. Every node is initially in an unassigned state. The root node takes the first step by assigning the level variable to the first element in the `Level` set and transitioning to the assigned state. Nodes that are in the assigned state can send a message to an arbitrary unassigned node. In Figure 6.6, an assigned node sends a message to a sister node that has not received a message. A message is a mapping from the identifier element of the node sending the message to its level. A node that has received a message assigns itself the next level in the order with respect to the level of the node that sent it the message and assigns its `parent` pointer (a variable) to the node that sent the message. Once a node has assigned its level and parent, it transitions to the assigned state and can send messages to nodes that are unassigned. We repeat the process of sending messages and assigning nodes until every node is in the assigned state.

[3]The complete model can be found in Appendix B.

Distributed Spanning Tree Algorithm					
Property	Snapshot Scope	Node Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Safety	9	4	495	1125	1171
Safety	11	5	479	2021	2591
Safety	13	6	1877	4855	11756
Liveness	9	4	130	n/a	606
Liveness	11	5	169	n/a	2673
Liveness	13	6	1504	n/a	7360

Table 6.2: Time Taken to Check Spanning Tree Properties (ms = milliseconds, n/a = not applicable)

In one big step, the nodes that have been assigned a level will send a message to unassigned nodes and unassigned nodes that have received a message will assign themselves a level and a parent. We repeat the big steps until every node has been assigned a level and a parent.

This model should satisfy some safety and liveness properties. The algorithm must not introduce a cycle in the network topology (safety) and every node must eventually be a part of the network topology (liveness). For the safety property, we check for cycles using the transitive closure operator over the parent relation. The property is shown in Figure 6.7 for the traces, TCMC, and Electrum model checking methods and specifies that no node except the root node is reachable from itself in the parent relation. For the liveness property, we check whether every node is in the assigned state at the end of the execution of the algorithm to determine if every node eventually becomes a part of the network topology. Table 6.2 shows the time taken to analyze each of the properties in the Distributed Spanning Tree algorithm. As the scope of the nodes is increased, the number of snapshots needed to compute the spanning tree is increased since we need to take more steps to ensure that every node is eventually assigned. Model checking using the traces method proved to provide the fastest analysis times.

The Dash+ features that are highlighted using this case study are:

1) *Communication between arbitrary sister components*: The Distributed Spanning Tree model in Dash+ has each copy of the replicated AND-state `N` accessing the `message` variable in a sister copy and adding a message to the `message` variable based on whether the sister copy has received a message. If we were to model the Distributed Spanning Tree algorithm in PlusCal or SPIN, we would need global channels (or global sequences in PlusCal) to which an assigned node can send a message. An unassigned node would

```
conc state DistrubedTreeSpanning {
  root: one Node

  conc state N [Node] {
    level: lone Level
    parent: lone Node
    message: Node -> Level

    action sendMessage [
      ...
    ] {}

    default state Unassigned {
      trans RootAssign {
        when {
          this in root
        }
        do {
          level' = nodeLevel/first
          parent' = this
        }
        goto Assigned
      }

      trans NodeAssign {
        ...
      }
    }

    state Assigned {
      trans sendMessage {
        ...
      }
    }
    ...
  }
}
```

Figure 6.5: Distributed Spanning Tree Algorithm in Dash+

```
action sendMessage [
  one n: Node - this | {
    // The sister Node has not received a message
    no N[n]/message
    // Send the sister Node a message with the
    // local Node's identifier and Level
    N[n]/message' = this -> level
    all others: Node - n | N[others]/message' = N[others]/
 message
  }
] {}
```

Figure 6.6: Sending a Message to a Sister Node

read the global buffer and use the information in the buffer to assign a parent and a level and remove the message from the buffer. Dash+ allows us to directly access variables in a sister node without having to resort to global variables and thus simplifying the model.

2) *Arranging replicated components in a tree structure*: The `parent` variable in the replicated AND-state `N` is defined as a mapping from a node to a parent node. Since the parent variable will eventually store a parent-child mapping for every node, we will have arranged the nodes in a tree structure. In this case, we do not constrain the identifier set node to be a tree, but we check that the algorithm produces a tree structure using the `parent` variable.

3) *Ease of checking properties*: We check for graph properties of the Distributed Spanning Tree model (such as whether a node is reachable itself) and checking for graph properties is easier in a relational language like Alloy.

```
//DistrubedTreeSpanning_root is the root Node
//DistrubedTreeSpanning_N_parent is the parent relation

// Using Traces
assert noCycles {
  (all s: Snapshot |
    no n: Node -  s.DistrubedTreeSpanning_root |
      // A Node is not reachable from
      // itself in the parent relation
      n in n.(^(~(s.DistrubedTreeSpanning_N_parent))))
}

// Using TCMC
assert ctl_noCycles  {
  ctl_mc[ag[{ s: Snapshot |
    all n: Node -  s.DistrubedTreeSpanning_root |
      n !in n.(^(~(s.DistrubedTreeSpanning_N_parent)))}]]
}

// Using Electrum
assert noCycles {
  always (one s: Snapshot |
    all n: Node -  s.DistrubedTreeSpanning_root |
    n !in n.(^(~(s.DistrubedTreeSpanning_N_parent))))
}
```

Figure 6.7: Checking for Cycles in the Distributed Spanning Tree

6.3 Leader Election Protocol

The Leader Election protocol [10] is used to select a leader in a network with processes arranged in a ring where the leader is a process with the highest identifier. Each process passes its identifier as a token in a specified direction around the ring. A process that receives a token from a neighboring process can either consume it or pass the token to the process next to it in the ring. A token is consumed by a process if the identifier in the token is less than the identifier of the process; a token is passed if the identifier in the token is greater than the identifier of the process. A process that receives a token with its own identifier will set itself as the election leader since a process with the highest identifier will only receive its token if it has been passed completely around the ring.

We have modelled the Leader Election protocol in Dash+ using a replicated AND-state `Process` parameterized by the `Identifier` set. The `Identifier` set is constrained using the ring module in order to arrange the processes in a ring. A snippet of the Leader Election protocol in Dash+ is shown in Figure 6.8. The replicated AND-state `Process` has a buffer variable that stores the tokens it receives. Every copy of `Process` is initially in an electing state and can either consume a token in its buffer or add the token (pass it along) to the buffer in the successor process. A token is consumed by removing it from the buffer. Once a process notices its own identifier in the buffer, it will transition to the elected state to indicate that it is the leader.

In one big step, every copy of `Process` may consume a token or pass it along. The model will keep on taking big steps until a leader is elected since there are no more small steps that can be taken once a leader is elected.

The Leader Election protocol was modelled by Jackson in Alloy [26] where each process is given a pool of tokens that can be consumed by a process or passed to a successor process. The token to pass or consume is selected arbitrarily from the pool of tokens. Jackson states that the Leader Election protocol should ideally work with buffered tokens. Therefore, we use the `token` buffer in our model that stores an ordered set of tokens such that each process can attend to tokens it has received in a FIFO manner.

The properties that we need to check of the leader election protocol is that one leader is eventually selected (liveness), and that the leader selected has the highest identifier (safety). We can check the liveness property by specifying that a copy of `Process` will always eventually be in the `Elected` variable. The safety property can be checked by specifying that an elected leader is last in the ordered `Identifier` set. Figure 6.9 shows the safety and liveness properties of the Leader Election Protocol for the Traces method. The time taken to run each of the properties using a maximum buffer size of 3 is shown

```
open util/ring [Identifier]
conc state System {
  elected: set Identifier

  conc state Process [Identifier] {
    token: buf[Identifier]

    default state Electing {
      trans PassToken {
        when { ... }
        do {
           // Pass the token
          Process[next]/token.addFirst[token.firstElem]
          ...
        }
      }

      trans ConsumeToken {
        when {...}
        do token.removeFirst // Consume the token
      }

      trans ElectLeader {
        when token.firstElem = this
        do elected' =  this // Elect Leader
        goto Elected
      }
    }
    state Elected {}

    init {
      one token
      // Initially have each process send a token to a successor
      Process[next]/token.firstElem = this
    }
  }
}
```

Figure 6.8: Leader Election Protocol in Dash+

```
//Liveness Property
//System_elected is the "elected" variable
assert eventuallyLeaderElected {
  some s: Snapshot | one s.System_elected
}

// Safety Property
assert electedHasHighestIdentifier {
// If the elected variable has an Identifier element, then that
// element must be the last element in the Identifier ordering
    (all s: one Snapshot |
  one s.System_elected => (s.System_elected) in PO/last)
}
```

Figure 6.9: Leader Election Properties for the Traces method

in Table 6.3. The Traces method of model checking provided the best analysis times. The analysis times using Electrum drastically increase as the number of steps is increased.

The Dash+ features that are highlighted using this case study are:

1) *Locality of description*: Dash+ gives us the opportunity to model the Distributed Spanning Tree algorithm with respect to a single node. In the Leader Election model by Jackson [26], each dynamic variable of a process is modelled as a function from the global state and process identifier to the value which adds extra parameterization to the model.

2) *Arranging replicated components in a ring:* We use the `ring` module to constrain the `Identifier` set such that the set of processes in the Leader Election model is arranged in a ring.

3) *Ease of checking for properties:* Since the processes in the Leader Election algorithm model in Dash+ are arranged in a ring structure, it is easier to check for graph properties using Alloy.

4) *Communication using buffers:* Each copy of `Process` in the Leader Election Ring model in Dash+ communicates with sister components by adding a token to the buffer in the sister copy next to it in the ring. In SPIN or PlusCal, modelling the Leader Election Protocol would require us to define global variables for each copy of a process since one process is not able to access a local variable in another process. Otherwise, each process would need to broadcast a message to a global variable that is then accessed by another process. Either technique would increase the complexity of the model.

Leader Election Properties					
Property	Snapshot Scope	Process Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Safety	13	6	5726	5903	17561
Safety	15	7	54735	12336	138053
Liveness	13	6	5086	n/a	17115
Liveness	15	7	63950	n/a	75972

Table 6.3: Time Taken to Check Leader Election Properties (ms = milliseconds, n/a = not applicable)

6.4 The Bit Counter

The Bit Counter [18] consists of an ordered set of bits where the first bit represents the least significant bit and the last bit represents the most significant bit of a counter. During each tick of a clock, the counter increments by one. An increment in the counter is represented by a combination of bits toggling their state from 0 to 1. Once a bit has toggled its state back to 0 from 1, the next bit in the order will toggle its state. A two-bit counter has been modelled in Dash by Serna [41] where each bit is represented by a concurrent state. The initial clock tick is represented by an environmental event with the effects represented by internal events. Figure 6.10 shows a snippet of the bit counter in Dash (left). The concurrent AND state `Bit1` is the least significant bit and `Bit2` is the most significant bit.

In Dash+, we model an n-bit Counter using a replicated AND-state parameterized by an identifier set (called `Identifier`) that is constrained by Alloy's ordering module and is shown in Figure 6.10 (right). The first element in the ordered identifier set represents the least significant bit and the last element in the set represents the most significant bit. Therefore, we specify an upper bound for the number of bits we want in our counter by using the scope for the `Identifier` set. A variable called `current` is used to keep a track of the current bit that is toggling its state and is initially set to the least significant bit in the model. An environmental event toggles the state of the current bit which sends an event to the next bit in the ordering as in the transition seen in Figure 6.11. Once the next bit receives an event, it takes a transition to toggle its state and assigns itself as the current bit. We keep iterating through each bit until the last bit has toggled its state and sends an event `Done` to signify that every bit has toggled its state. In each big step, one bit may only toggle its state, or a bit will toggle its state and send an event to the next bit causing that bit to toggle its state within the same big step.

```
conc state Bit1 {
  env event Tk0 {}
  event Tk1 {}

  default state Bit11 {
    trans T1 {
      on Tk0
      goto Bit12
    }
  }
  state Bit12 {
     trans T2 {
      on Tk0
      goto Bit11
      send Tk1
    }
  }
}

conc state Bit2 {
  event Done {}
  default state Bit21 {
    trans T3 {
      on Bit1/Tk1
      goto Bit22
    }
  }
  state Bit22 {
    trans T4 {...}
  }
}
```

```
conc state Counter {
  env event Tk0 {}
  event Done {}
  current: one Identifier

  conc state Bit [Identifier]{
    event Tk1 {}

    default state Bit1 {
      trans currentBitToBit2{
        ...
      }
      trans nextBitToBit2 {
        ...
      }
    }
    state Bit2 {
      trans currentBitToBit1 {
        ...
        send Tk1[next[this]]
      }
      trans nextBitToBit1 {
        ...
        do current' =
          next[current]
        goto Bit1
      }
      trans lastBitDone {
        ...
        send Done
      }
    }
    ...
  }
}
```

Figure 6.10: Bit Counter model in Dash (left) and Dash+ (right)

```
trans currentBitToBit1 {
  // When the env event Tk0 is triggered
  on Tk0
  // this bit is currently toggling
  when this in current
  // Transition to the basic state Bit1
  goto Bit1
  // event Tk1 triggered in the next bit in the ordering
  // (next[this]) returns the next bit
  send Tk1[next[this]]
}
```

Figure 6.11: Sending an Event to a Sister Component

Bit Counter Properties					
Property	Snapshot Scope	Bit Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Every Bit Toggled	12	3	286	1226	787
Every Bit Toggled	20	4	295	4457	546

Table 6.4: Time Taken to Check Bit Counter Properties (ms = milliseconds)

The property that we need to check of the bit counter is that every bit has toggled its state once the next `Done` event has been sent. The time taken to analyze the properties is shown in Table 6.4. The Traces and Electrum method of model checking provides the best analysis times.

The Dash+ features that are highlighted using this case study are:

1) *Easier scalability:* The number of bits in the Bit Counter can be increased by incrementing the scope of the identifier set. In Dash, we need to describe an AND-state to represent each bit. Therefore, the Bit Counter model by Serna has two-bit and requires two AND-states (`Bit1` and `Bit2`) to represent each bit. If we were required to have three bits in the Bit Counter, the Dash model would need another AND-state to represent the third bit.

2) *Linear ordering:* The bits in the Bit Counter model in Dash+ are arranged in a linear order by constraining the identifier set using Alloy's ordering module. Therefore, the first bit is represented by the first identifier element in the order, and the last bit is represented by the last bit in the order.

3) *Communication by sending events to a sister component:* In Dash+, we trigger an event in the next bit within the ordering to specify that the next bit should toggle its state. The Bit Counter model in Dash requires us to trigger an event in the current bit after toggling its state, and having the next bit respond to the event triggered in the current bit.

6.5 Elevator

A model of an elevator system was designed by Serna [41] in Dash inspired by the elevator model in [19] and [38]. The elevator can move in an up or down direction to respond to a call from any floor and will stop once it reaches a floor from which a call is made. If there are no more calls in the direction of travel, the elevator will change its direction to respond to any calls in the opposite direction. Once the elevator has responded to every call that has been made, it will move to the ground floor and wait for further calls.

We extend the elevator model in Dash+ to consist of a set of elevators running concurrently and a controller that decides which elevator will get a call depending on their current floor and direction. Figure 6.12 shows a snippet of the Elevator model in Dash+ where the set of elevators is declared using the replicated AND-state `Elevator`; the controller is declared as a non-replicated AND-state. Each copy of the replicated AND-state `Elevator` is initially in the idle state meaning that it has no calls to respond to. The AND-state `Controller` initially has a set of calls with a specified floor and direction that elevators need to eventually respond to. During each big step, the controller will send a call to a copy of `Elevator` based on the direction of the elevator and its proximity to the floor calling it. Figure 6.13 shows the specification of an action for the `Controller` sending a request to an elevator to go to a floor above it. An elevator closest to the floor that is requesting a call and is traveling in the same direction as that of the requested call will receive the call. Since the `Controller` initially has a set of calls that it can send to the elevators, the controller will eventually be exhausted of calls to send and the elevators will have responded to every call that was sent.

The properties that we check of the elevator is that all the requests are eventually sent (the `Controller` initially has an arbitrary set of calls that it sends out over time) by

```
conc state System {
  conc state Controller {
    callToSend: Floor -> Direction

    // Send a request to an Elevator to move to a floor above it
    action SendUpRequest [...] {}
    // Send a request to an Elevator to move to a floor below it
    action SendDownRequest [...] {}

    default state Sending {
      trans SendingUpRequest {
        ...
      }
      trans SendingDownRequest {
        ...
      }
    }
  }

  conc state Elevator [Identifier] {
    direction: one Direction
    called: set Floor
    current: one Floor

    state MovingUp {...}
    state MovingDown {...}

    default state Idle {...}
  }

  init {
    no called
    current = min[Floor]
    direction = Up
  }
}
```

Figure 6.12: Elevator in Dash+

```
action SendUpRequest [
  one e0: Identifier, f: callToSend.Up | {
    // The direction of the Elevator is "Up"
    Elevator[e0]/direction = Up
    // The elevator is below the floor that it needs to go to
    lte[Elevator[e0]/current, f]
    // Send the call to the elevator
    Elevator[e0]/called' = Elevator[e0]/called + f
    callToSend' = callToSend - (f -> Up)
    // Elevator getting the call is the closest
    no e1: Identifier - e0 | { Elevator[e1]/direction = Up and
      between[Elevator[e0]/current, Elevator[e1]/current, f] }
    all others: Identifier - e0 | {
      Elevator[others]/called' = Elevator[others]/called }
  }
] {}
```

Figure 6.13: Sending a Call to an Elevator

the `Controller` (liveness) and that an elevator that is called is always the one closest in proximity to the floor calling it (there are no more elevators that are traveling in the same direction as the elevator that is called and is closer in proximity to the floor calling the elevator). The time taken to analyze the properties is shown in Table 6.5.

The Dash+ features that are highlighted using this case study are:

1) *Replicated and non-replicated components running concurrently*: The Elevator model in Dash+ has a non-replicated AND-state (`Controller`) and a replicated AND-state (`Elevator`) that run concurrently together. The replicated AND-state `Elevator` changes its state based on the calls sent by the `Controller`.

2) *Communication between replicated and non-replicated components*: The non-replicated AND-state `Controller` is able to observe the state of each of the replicated AND-state `Elevator` and send a message to a copy of `Elevator` (by accessing a variable) based on the floor and direction of the elevator.

Elevator Properties					
Property	Snapshot Scope	Elevator Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Safety	17	3	627	13446	34017
Safety	17	4	878	16325	106855
Liveness	17	3	1393	n/a	9850
Liveness	17	4	1306	n/a	20537

Table 6.5: Time Taken to Check Elevator Properties (ms = milliseconds, n/a = not applicable)

6.6 Carousel

Carousel [46] is a distributed system that aims to provide low-latency transaction processing for multi-partition geo-distributed transactions. It uses a two-phase commit protocol to ensure that transactions are committed atomically and a consensus protocol is used for fault tolerance. A **two-phase commit** (2PC) protocol is an atomic commitment protocol that coordinates processes taking part in a distributed atomic transaction to determine if a transaction should be committed or aborted. The **consensus protocol** replicates updates to a database to multiple data centers to ensure that a distributed system can tolerate a specified maximum number of faults. Distributed systems such as Spanner and CockroachDB use the 2PC and consensus protocol to commit transactions to data partitions, but these protocols are carried out sequentially meaning the time to complete transactions is significantly increased. Carousel parallelizes the 2PC and consensus protocol steps to achieve reduced transaction completion time.

Carousel replicates data partitions and stores the data partitions across different data centers for fault tolerance. Replicas of a partition form a consensus group. A client can initiate a transaction to write read and/or write data to a participant partition and the leader of a participant partition's consensus group is called the participant leader. The participant leader is responsible for writing data to a partition once a decision to commit has been made and one participant leader will act as a coordinator to decide whether a transaction will be committed or aborted. The steps taken to commit a transaction are shown in Figure 6.14. A client sends a transaction request to a coordinator and participant leaders with key-value pairs that need to be updated in a partition. A coordinator will receive the request, replicate the data and wait for a prepared message from all the

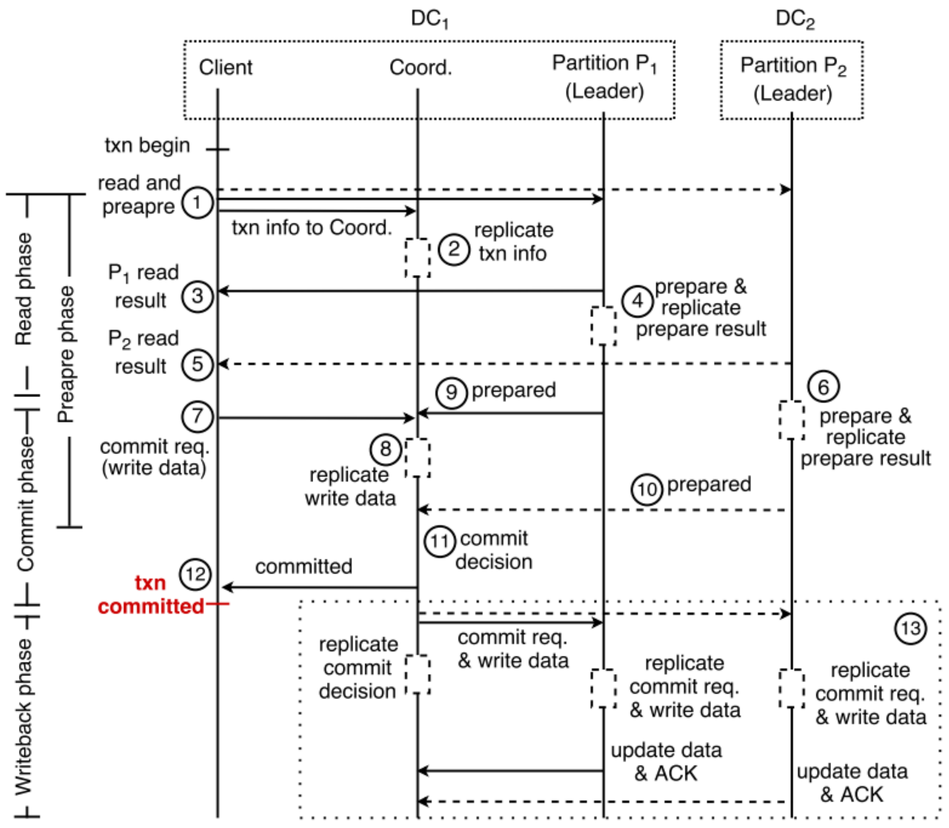

Figure 6.14: A Basic Version of Carousel [46]

participant leaders meaning that a transaction can be committed by the partitions. Once a coordinator receives a prepared request from every participant leader, the coordinator will send commit messages to the client and the participant leaders to indicate that the transaction can be committed. Participant leaders that receive the commit message will update their data and complete a transaction.

We have modelled Carousel in Dash+ where clients, coordinators, and participant leaders are described as replicated AND-states. Each copy of the `Client` AND-state can send a transaction request to every copy of the replicated AND-state `Participant Leader` and one copy of the replicated AND-state `Coordinator`. A transaction is defined as a signature with a relation mapping to a key-value tuple. A participant leader that receives a transaction from a client will store the transaction in a buffer until the transaction has either been committed or aborted. A transaction is aborted if the key-value pair that needs to be updated by a transaction conflicts with a pending transaction in a participant leader. The pending transactions for a participant leader is stored in a buffer variable. Otherwise, the participant leader will inform a copy of the replicated AND-state `Coordinator` managing the transaction that the transaction has been prepared and can be committed. Once the `Coordinator` receives a prepared response from every copy of `Participant Leader`, the coordinator sends a commit message to the participant leaders and the client which requested the transaction meaning that the transaction can be safely committed without any conflicts. Participant leaders that receive the commit message from a coordinator will update their data with the key-value pair in the transaction that has been committed. A statecharts-like diagram of the Dash+ model of Carousel is shown in Figure 6.15.

The properties that we check of the Carousel model in Dash+ are:

1. *Client receives acknowledgment*: A client receives an acknowledgment of a commit if all participant leaders commit a transaction

2. *Transaction aborted*: A transaction is always aborted if one or more participant leaders abort a transaction

3. *No comment if a participant leader aborts*: A participant leader cannot commit a transaction if sister participant leaders have aborted the transaction

4. *Commit updates data*: If a transaction is committed, every participant leader must update their key-value pairs

The time taken to analyze the properties for Carousel are shown in Table 6.6. The snapshot scope that we have chosen for the properties are sufficient enough for the model to complete at least two transactions or at most three transactions depending on whether transactions are committed or aborted. We have a higher snapshot scope for checking properties with more participant leaders as it will take more time to complete a transaction with more participant leaders. We also want to ensure that the snapshot scope is large enough to satisfy the significance axiom such that every transition in the model can be taken within the snapshot scope specified.

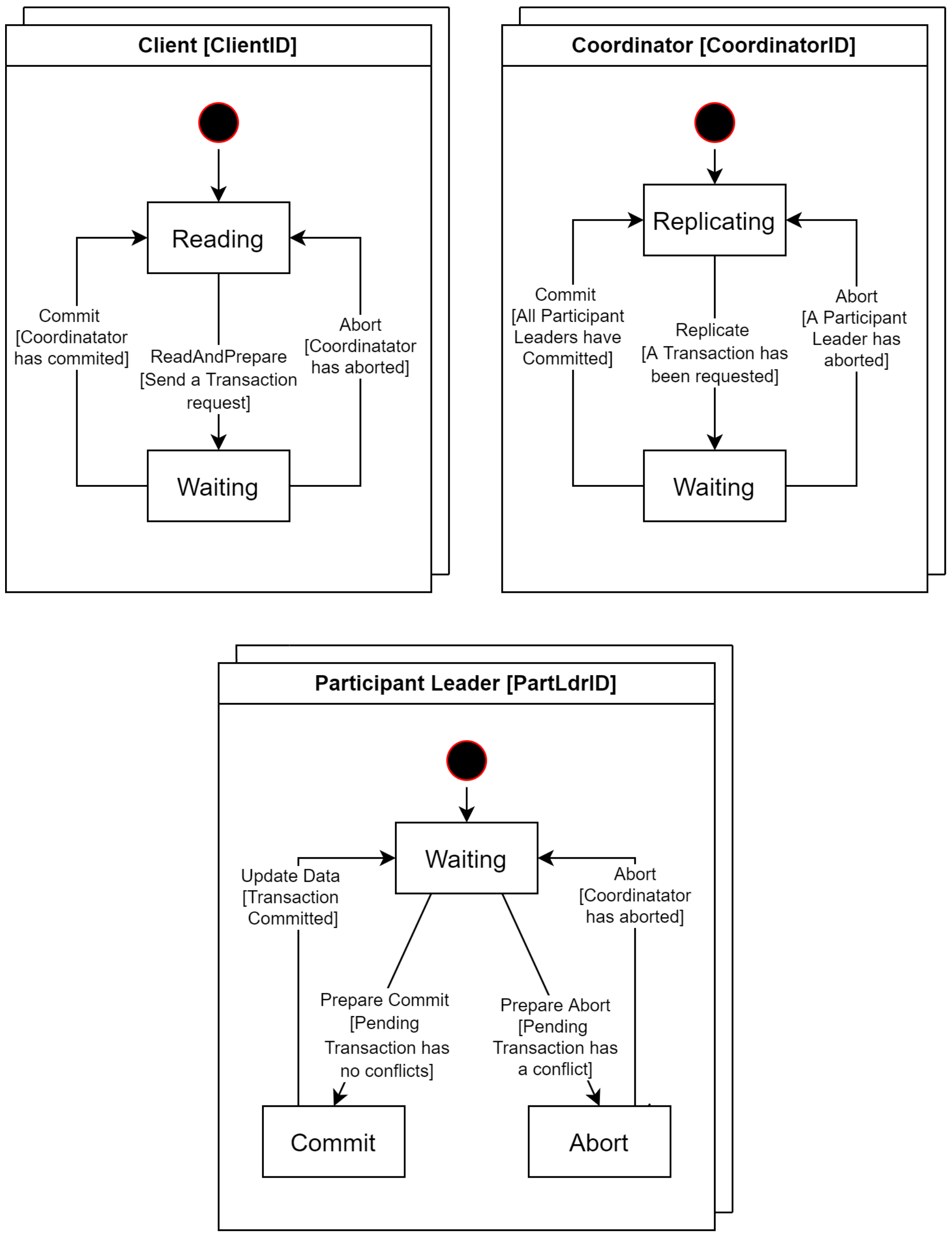
Client [ClientID]
Reading
Commit [Coordinatator has commited]
ReadAndPrepare [Send a Transaction request]
Abort [Coordinatator has aborted]
Waiting
Coordinator [CoordinatorID]
Replicating
Commit [All Participant Leaders have Committed]
Replicate [A Transaction has been requested]
Abort [A Participant Leader has aborted]
Waiting
Participant Leader [PartLdrID]
Waiting
Update Data [Transaction Committed]
Abort [Coordinatator has aborted]
Prepare Commit [Pending Transaction has no conflicts]
Prepare Abort [Pending Transaction has a conflict]
Commit
Abort

Figure 6.15: Carousel in Dash+

Carousel Properties							
Property	Snapshot Scope	Client Scope	Coord Scope	PL Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Client receives ack	23	2	2	2	4153	n/a	3924
Client receives ack	30	2	2	3	5897	n/a	4090
Transaction aborted	23	2	2	2	3707	n/a	3299
Transaction aborted	30	2	2	3	5889	n/a	4082
No commit if PL aborts	23	2	2	2	3793	25818	3294
No commit if PL aborts	30	2	2	3	5888	51448	4079
Commit updates data	23	2	2	2	3710	n/a	3304
Commit updates data	30	2	2	3	5890	n/a	4082

Table 6.6: Time Taken to Check Carousel Properties (ms = milliseconds, n/a = not applicable, PL = participant leader, Coord = coordinator)

The Dash+ features that are highlighted using this case study are:

1) *Multiple replicated components running concurrently*: The Carousel model in Dash+ has multiple replicated AND-states running concurrently that change their state based on the state of other replicated AND-states.

2) *Communication between replicated components*: A copy of the replicated AND-state `Client` can add a transaction to the buffer in every copy of the replicated AND-state `Participant Leader`. Each copy of `Participant Leader` can update a variable in a copy of the replicated AND-state `Coordinator` which in turn can update a variable in every copy of `Participant Leader` to signify whether a transaction is committed or aborted. Therefore, Dash+ gives us the opportunity to have a copy of a replicated component communicate with one or more copies of another replicated component and update their variables or buffers.

6.7 Heating System

The Heating System [17] consists of a set of rooms, a controller, and a furnace that run concurrently. Each room has a desired temperature, an actual temperature, and a valve position. If the actual temperature of a room falls below the desired temperature, the value position is adjusted and the room waits for a change in the actual temperature. If the desired effect is not achieved within a period of time, the room will make a request to the controller for heat. The controller is responsible for turning on the furnace once one or more rooms request for heat, and turning off the furnace if no rooms are requesting for heat. Faults can occur in the furnace at any time and the controller is informed of any faults in the furnace which causes the system to transition to an `Error` state. The furnace will be reset when the controller has been reset by a user.

We modelled the heating system in Dash+ using two top-level OR states called `Functioning` and `Error`[4]. The `Functioning` OR-state has two non-replicated AND-states `Controller` and `Furnace`, and a replicated AND-state `Room` as seen in Figure 6.16. Each copy of the replicated AND-state `Room` has variables to keep track of the actual temperature, desired temperature, and valve position. If the actual temperature is below the desired temperature for a copy of `Room`, a boolean variable `requestHeat` is set to True meaning that the room is requesting heat from the furnace. The AND-state `Controller` checks on whether any copy of `Room` has requested heat and triggers an `activate` event if one or more copies of `Room` has made a request for heat. Once the `activate` event has

[4]The complete model of the Heating System in Dash+ can be found in Appendix C.

been triggered, the AND-state `Furnace` will transition to the `Furnace_Running` state meaning that the furnace has been activated and is now providing heat to the rooms. We model a fault in the furnace using an environmental event `furnaceFault` which causes the system to transition to the `Error` state from the `Functioning` state. Since every concurrent state is declared within the `Functioning` OR-state, the transition to the `Error` state will result in the concurrent states leaving their current active basic state.

Our model differs slightly from the model of the heating system in [17]. The model of the heating system in [17] has an `Error` state declared within the AND-states `Controller` and `Furnace`, but the Dash+ model instead transitions into an `Error` state to indicate that the furnace has a fault. We have made the change to abstract the common transitions in [17] since we can do so by nesting replicated AND-states and non-replicated AND-states within an OR-state. The system transitions to the `Functioning` state on an environmental event causing the AND-state states declared within the `Functioning` OR-state to enter their default states.

We check a liveness and safety property of the heating system. The liveness property specifies that the heater eventually turns on if a room requires heat; the safety property specifies that rooms will always receive heat if the actual temperature of the room falls below the desired temperature. The time taken to analyze the properties is shown in Table 6.7.

The Dash+ features that are highlighted using this case study are:

1) *Multiple replicated components running concurrently*: The Heating System model in Dash+ has multiple AND-states (`Controller` and `Furnace`) running concurrently with a replicated AND-state (`Room`).

2) *Communication between replicated and non-replicated AND-states*: In the Heating System, the non-replicated component `Controller` observes the values of variables in each copy of the replicated AND-state room to determine if a room needs heat. The AND-states `Controller` and `Furnace` both communicate with each other and the error state by triggering events declared globally.

3) *Nesting of a replicated and non-replicated AND-state*: The replicated AND-state `Room` and the non-replicated AND-states `Controller` and `Furnace` is nested within an OR-state `Functioning` which is nested inside a non-replicated AND-state. As a result, exiting the OR-state `Functioning` means that any nested AND-states will exit their active state.

```
sig Temp {}
conc state HeatingSystem {
  ...
  default state Functioning {
    conc state Furnace {
      default state Furnace_Normal {
        default state Furnace_Off {...}
        state Furnace_Running {...}
      }
    }
    conc state Controller {
      condition heatRequested [...] {}
      condition noHeatRequested [...] {}
      default state Off {...}
      state On {
        default state Idle {...}
        state Heater_Active {...}
        ...
      }
    }
    conc state Room [Identifier] {
      actualTemp: one Temp
      desiredTemp: one Temp
      valvePos: one ValvePos
      requestHeat: one Bool
      ...

      default state No_Heat_Request {
        default state Idle_No_Heat {...}
        state Wait_For_Heat {...}
      }
      state Heat_Requested {...}
      init {...}
    }
  }
  state Error {...}
}
```

Figure 6.16: Heating System in Dash+

Heating System Properties					
Property	Snapshot Scope	Room Scope	Traces (ms)	TCMC (ms)	Electrum (ms)
Safety	25	2	2085	37131	4678
Safety	30	3	2710	71644	6091
Liveness	25	2	15798	n/a	22457
Liveness	30	3	491878	n/a	52932

Table 6.7: Time Taken to Check Heating System Properties (ms = milliseconds, n/a = not applicable)

6.8 Summary

We evaluate Dash+ using case studies that exploit the features in Dash+. Each of the case studies covers one or more specific features in Dash+ and we try to ensure that every feature in Dash+ has been covered by the case studies. We also evaluate the performance in analyzing Dash+ models using the model-checking options available in Dash+.

In Dash+, we can arrange replicated components in a specific topology (such as a ring or linear structure) using modules or constraints on the identifier signature. As a result, modelling Chord or the Bit Counter is convenient as we can arrange the replicated components in the required topology by using the built-in modules that are available to the modeller. Furthermore, we are able to model processes with a local point of view but we can use Alloy's rich logic to specify graph properties for case studies such as Chord, and the Distributed Spanning Tree. The control state hierarchy in Dash+ also means that we can use the control state hierarchy to understand the state of a process. For example, we can check whether a node is active or failed in Chord depending on its current basic and not have to resort to extra variables.

In our case studies, we found that it is convenient to model distributed systems using Dash+ as there are constructs for direct communication between sister components and multiple replicated AND-states. Distributed systems such as Chord and Carousel have replicated components running concurrently and communicating with each other. The constructs in Dash+ for direct communication between replicated components make it easier to model the distributed systems by not having to resort to global variables. As we do not use any global variables for communication, the complexity of the model is decreased.

In analyzing our case studies, the Traces method provided the best performance with respect to analysis times in a majority of the cases. The performance of model checking using Electrum can vary depending on the complexity of the model and the number of copies of replicated components. Models that are more complex and use more copies of replicated components will take a much longer time to check for properties using Electrum. For model checking using TCMC, the performance drastically increases as the number of snapshots is increased.

Chapter 7

Related Work

In this chapter, we make a comparison between Dash+ and several formal modelling languages. We point out the advantages that Dash+ offers over each of these languages.

7.1 Languages with Constructs for Processes

PlusCal [36] is a formal specification language used for writing algorithms with concurrent processes. Processes in PlusCal communicate with each other by using variables and sequences declared globally. A sequence is an ordered set of elements with constructs for appending an element, accessing an element from the head or tail, and fetching the length of the ordered set. PlusCal expressions are written in the TLA+ (the Temporal Logic of Actions) [33] language and it is a high-level specification language based on first-order logic and set theory. TLA+ is used for describing the behavior of concurrent and distributed systems. PlusCal is translated to a TLA+ specification using the TLC model checker [35].

Promela (Process or Protocol Meta Language) [24] is a verification modeling language used for designing and verifying asynchronous process systems used with the SPIN model checker. It has an emphasis on creating abstractions of concurrent software systems and verifying the behavior of clients and servers in networks of processors. A model written in Promela is comprised of a set of processes. A process can be replicated by specifying the number of copies to create in a parameter within the model. Each process can have its own variables or channels (buffers) that can be locally accessed. Processes interact with each other using globally declared variables and/or synchronous or asynchronous channels.

Dash+ offers an advantage over PlusCal and Promela with its state hierarchy. There is no notion of state hierarchy in PlusCal and Promela and we cannot declare processes within processes. Furthermore, processes in Dash+ can directly exchange information by accessing variables in sister components. The only means for processes to communicate in PlusCal and Promela is through the use of variables that are declared globally and increases the complexity of PlusCal and Promela models that require processes to directly communicate with other. In Promela and PlusCal, we can arrange processes in a topology by giving a process an identifier using global variables/buffers and defining functions that specifies how each process identifier relates to another process identifier (as seen in [48]). The buffer structure in Dash+ also provides an advantage over the sequences in PlusCal as a buffer in Dash+ has constructs for appending or removing an element to/from the head or tail. There are no constructs for removing an element from a sequence or appending an element to the head of a sequence in PlusCal. One advantage that Dash+ provides over Promela is the ease of writing graph properties. The properties in Promela are written in C and it can be difficult to write graph properties in C [47]. Promela also has limited datatypes and data operations because it focuses on the communication and synchronization aspects of a model. However, Promela does hold one advantage over Dash+ as processes can be dynamically created during analysis which is a feature that Dash+ lacks.

7.2 Declarative Languages and Languages Based on Alloy

Electrum [7] is an extension to Alloy for modelling transition systems that includes the keyword "var" as syntax to denote the declaration of dynamic elements of a model. DynAlloy [21] 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 precondition and postcondition to describe changes to the system after the action and are strongly influenced by programming language constructs [41]. Both of these extensions lack state hierarchy, replicated concurrent states, and communication using buffers, which Dash+ provides.

Declarative modelling languages such as B [1], TLA+ [15], VDM [28], and Z [43], are based on first-order logic and/or set theory abstractions to formally describe systems with complex structures. They describe changes to a system using primed expressions. However, they lack the notion of a state hierarchy and replicated components that Dash+ is able to provide.

7.3 Languages with State Hierarchy

UML [40] state machines support hierarchical and concurrent labelled control state hierarchy. Through the use of object modelling, UML supports replicated concurrent components. Using OCL [9], pre- and postconditions 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.

Chapter 8

Conclusion

This thesis presents Dash+, an extension to Dash for modelling transition systems with replicated components. We provide Dash+ with language constructs for describing a model with replicated concurrent components and allow communication among these components and between these components. 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. A model in 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 are 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 have also provided Dash+ with three methods for model checking: traces-based model checking, transitive-closure-based model checking, and Electrum. The modeller can choose to use any of these model checking options for model checking Dash+ models.

8.1 Future Work

There are several avenues for improving Dash+ in the future:

Failure State: Dash+ might benefit from having explicit language constructs for declaring a failure state. A construct for modelling the failure of a concurrent state would simplify the design of models in which a concurrent system might fail at any time. In the real-world scenario, it is expected that servers in distributed systems can fail at any moment and distributed systems such as Carousel [46] account for an expected number of server failures. One way to model a failure would be to use a state that an AND-state can transition to in the event of a failure. Any AND-states that are in a failure state would not be able to take transitions until they exit the failure state.

Optimizations for Analysis Time: It would help reduce the analysis time for Dash+ models by optimizing the translation process to exclude any unused parts of the Alloy model for particular properties. Furthermore, it might be beneficial to consider whether having processes specified locally can help facilitate symmetry breaking at the process level

Supporting Dynamic Process Creation: In Promela, it is possible to dynamically create processes during the execution of a model. It would be interesting to consider such a feature for Dash+ in which we can dynamically create processes during execution.

Visualization of Instances: The visualization of instances in Dash+ is unchanged when compared to the visualization of instances in Dash. Since Dash+ makes use of multiple `conf` and `taken` sets, it can be difficult to understand the active states and transitions taken in a big step so far. The modeller has to track the tuples in every `conf` and `taken` relation and how each relation changes its tuples after each small step. Simplifying the visualization (such as by making a change such that all the `conf` and `taken` relations are condensed) would greatly assist in understanding the instances.

It would also be interesting to consider analyzing Dash+ models as parameterized systems with unbounded values for the parameter as was done in the Leader Election Protocol in [4].

References

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996.

[2] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 09 1987.

[3] Amin Bandali. *A Comprehensive Study of Declarative Modelling Languages.* MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2020.

[4] Rylo Ashmore, Arie Gurfinkel, and Richard J. Trefler. Local reasoning for parameterized first order protocols. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2019.

[5] Mordechai Ben-Ari. *Principles of the SPIN model checker.* Springer, 2008.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117 – 148, 12 2003.

[7] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. The Electrum analyzer: model checking relational first-order temporal specifications. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 884–887. ACM, 2018.

[8] Julien Brunel, David Chemouil, and Jeanne Tawa. Analyzing the fundamental liveness property of the chord protocol. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[9] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): A definitive guide. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer, 2012.

[10] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, 1979.

[11] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, 1979.

[12] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *Int. J. Softw. Tools Technol. Transf.*, 2(4):410–425, 2000.

[13] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[14] Renato Mascarenhas Costa. *Compiling distributed system specifications into implementations.* PhD thesis, University of British Columbia, 2019.

[15] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154. Springer, 2012.

[16] Alcino Cunha. Bounded model checking of temporal formulas with Alloy. In *Lecture Notes in Computer Science*, pages 303–308. Springer Berlin Heidelberg, 2014.

[17] Nancy A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis.* PhD thesis, University of British Columbia, Department of Computer Science, October 1998.

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

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

[20] Sabria Farheen, Nancy A. Day, Amirhossein Vakili, and Ali Abbassi. Transitive-closure-based model checking in Alloy. *Journal of Software and Systems Modelling*, 19:721–740, 2020.

[21] Marcelo Frias, Juan Pablo Galeotti, Carlos Lopez Pombo, and Nazareno Aguirre. Dynalloy: Upgrading Alloy with actions. *Proceedings - 27th International Conference on Software Engineering, ICSE05*, pages 442–451, 01 2005.

[22] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[23] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

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

[25] Tamjid Hossain and Nancy A. Day. Dash+: Extending alloy with hierarchical states and replicated processes for modelling transition systems. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 21–29, 2021.

[26] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[27] Daniel Jackson. Software abstractions: Logic, language, and analysis. *Journal of Functional Programming*, 19:253–254, 03 2009.

[28] Clifford B. Jones. *Systematic software development using VDM (2. ed.).* Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[29] Magdalena Kacprzak, Alessio Lomuscio, and Wojciech Penczek. Bounded versus unbounded model checking for interpreted systems. 01 2004.

[30] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal verification of requirements using SPIN: A case study on web services. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 406–415. IEEE Computer Society, 2004.

[31] Mitchell Kember, Lynn Tran, George Gao, and Nancy A. Day. Extracting counterexamples from transitive-closure-based model checking. In Marsha Chechik, Daniel Strüber, and Dániel Varró, editors, *Proceedings of the 11th International Workshop on Modelling in Software Engineerings, MiSE@ICSE 2019, Montreal, QC, Canada, May 26-27, 2019*, pages 47–54. ACM, 2019.

[32] D. Richard Kuhn, Ramaswamy Chandramouli, and R Butler. Cost effective use of formal methods in verification and validation foundations. 02 V&V Workshop, Laurel, MD, USA, 2002-10-01 00:10:00 2002.

[33] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ toolbox. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 50–62, 2019.

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

[35] Leslie Lamport. The Pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[36] Leslie Lamport. *A PlusCal User's Manual (C-Syntax)*, version 1.8 edition, August 2018.

[37] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 373–383. ACM, 2016.

[38] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[39] Óscar R. Ribeiro, João M. Fernandes, and Luís F. Pinto. Model checking embedded systems with PROMELA. In *12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 April 2005, Greenbelt, MD, USA*, pages 378–385. IEEE Computer Society, 2005.

[40] Bran Selic, Conrad Bock, Steve Cook, Pete Rivett, Tom Rutt, Ed Seidewitz, and Doug Tolbert. OMG unified modeling language (version 2.5), 03 2015.

[41] Jose Serna. *Dash: Declarative Behavioural Modelling in Alloy.* MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2019.

[42] I. Shlyakhter, M. Sridharan, and D. Jackson. Analyzing distributed systems with first-order logic. 2002.

[43] John Michael Spivey. *The Z notation - a reference manual.* Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[44] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, page 149–160, New York, NY, USA, 2001. Association for Computing Machinery.

[45] Hillel Wayne. *Practical TLA+: Planning Driven Development.* Apress, 2018.

[46] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 231–243. ACM, 2018.

[47] Pamela Zave. Using lightweight modeling to understand chord. *Comput. Commun. Rev.*, 42(2):49–57, 2012.

[48] Pamela Zave. A practical comparison of Alloy and SPIN. *Formal Aspects Comput.*, 27(2):239–253, 2015.

APPENDICES

Appendix A

Chord Model

```
open util/ring[Node] as node

abstract sig Status {}
one sig Stabilizing, Rectifying extends Status{}

pred between [n1, nb, n2: Node] {
      lt[n1,n2] =>   ( lt[n1,nb] && lt[nb,n2] )
            else ( lt[n1,nb] || lt[nb,n2] )  }

// Returns the first node if current node is the last Node in
   the ordering
fun nextNode [n: Node] : lone Node  {
  {m: Node | no node/next[n] implies m = node/first
    else m = node/next[n]
  }
}

// Returns the previous node in the ordering
fun prevNode [n: Node] : lone Node  {
  {m: Node | no node/prev[n] implies m = node/last
    else m = node/prev[n]
  }
}

conc state System {
  members: Node
```

```

conc state N [Node] {
  frst: lone Node
  scnd: lone Node
  prdc: lone Node
  status: Status
  saved: lone Node

  default state Live {
    trans Fail {
      when {
        // We cannot fail if it would leave a
        // member with no successors
        all n: Node - this |
          some (members - this) &
          (Node[n]/frst + Node[n]/scnd)
      }
      do {
        members' = members - this
        frst' = {none}
        scnd' = {none}
        prdc' = {none}
        status' = {none}
      }
      goto Failed
    }

    trans StabilizeFromSucc {
      when {
        no status
      }
      do {
        // The Sucessor is dead
        frst !in members implies {
          frst' = scnd
          scnd' = nextNode[scnd]
          all n: Node | N[n]/status' = N[n]/status
          all n: Node | N[n]/saved' = N[n]/saved
        // The sucessor is a member
        } else {
          frst' = frst
          scnd' = N[frst]/frst
          // The sucessor's predecessor is better
          (some N[frst]/prdc
            and between[this, N[frst]/prdc, frst]
            and (N[frst]/prdc in members)) => {
            status' = Stabilizing
            saved' = N[frst]/prdc
            all n: Node - this | N[n]/status' = N[n]/status
            all n: Node - this | N[n]/saved' = N[n]/saved
          } else (this !in N[frst]/prdc) => {
            N[frst]/status' = Rectifying
            N[frst]/saved' = this
            all n: Node - frst | N[n]/status' = N[n]/status
            all n: Node - frst | N[n]/saved' = N[n]/saved
          } else {
            all n: Node | N[n]/status' = N[n]/status
            all n: Node | N[n]/saved' = N[n]/saved
          }
        }
      }
    }

    trans StabilizeFromPrdc {
      when {
        this in members
        status = Stabilizing
        between[this, saved, frst]
      }
      do {
        no status'
        no saved'
        // The Successor's Predecessor is dead
        saved !in members implies {
          frst' = frst
          scnd' = scnd
          all n: Node - this | {
            N[n]/status' = N[n]/status
            N[n]/saved' = N[n]/saved }
        // The Successor's Predecessor is a member
        } else {
          frst' = saved
          scnd' = N[saved]/frst
          N[saved]/status' = Rectifying
          N[saved]/saved' = this
          all n: Node - saved - this | {
            N[n]/status' = N[n]/status }
        }
      }
    }

    trans Rectify {
      when {
        this in members
        status = Rectifying
      }
      do {
        (between[prdc, saved, this] or
        prdc !in members or no prdc) => {prdc' = saved}
        else {
          prdc' = prdc
        }
      }
    }

    trans Flush {
      when {
        this in members
        prdc !in members
      }
      do {
        prdc' = {none}
      }
    }
  }

  state Failed {
    trans Join {
      when {
        Node in members
        }
        do {
          members' = members + this
        }
        goto Live
      }
    }

    init {
      no status
      no saved
      frst = nextNode[this]
      scnd = nextNode[frst]
      prdc = prevNode[this]
    }
  }

  init {
    // All Nodes are members initially
    Node in members
  }
}

fact alwaysThreeMembers {
  #(System_members) >= 3
}

/** FUNCTIONS FOR CHECKING PROPERTIES **/

// Returns a tuple (Node, Node) where the second element is a Live successor of the first element
fun succ : Node -> Node {
  { m1, m2: Variables.System_members |
     m1.(System_N_frst) in Variables.System_members =>
    (m2 = m1.(System_N_frst)) else (m2 = m1.(System_N_scnd)) }
}

// Returns the members that form a ring
fun ring : some Node { {m: Variables.System_members | m in m.^(succ)} }

fun appendages: set Node { Variables.System_members - ring }

/** PROPERTIES **/

// Members form atleast one ring
pred atLeastOneRing  {
  some ring
}

// Members form atmost one ring
pred atMostOneRing {
  (all m1, m2: ring| m1 in m2.^(succ))
}

// Members are ordered
pred orderedRing {
  all disj m1, m2, mb: ring |
    m2 = m1.^(succ) implies not between[m1, mb, m2]
}

// Member successors are ordered
pred orderedSuccessors {
  (all m: Variables.System_members |
    between[m, m.(System_N_frst), m.(System_N_scnd)])
}

pred connectedAppendages {
  (all m1: appendages | some m2: ring | m2 in m1.^(succ))
}

pred valid {
  atLeastOneRing
  atMostOneRing
  orderedSuccessors
  connectedAppendages
  orderedRing
}

pred ideal {
  valid
  no appendages
  (System_N_frst) = ~(System_N_prdc)
}
```

Appendix B

Distributed Spanning Tree Model

```
open util/traces[Level] as nodeLevel

sig Level {}

conc state DistrubedTreeSpanning {
  root: one Node

  conc state N [Node] {
    level: lone Level
    parent: lone Node
    message: Node -> Level

    action sendMessage [
      one n: Node - this | {
        no N[n]/message
        N[n]/message' = this -> level
        all others: Node - n |
          N[others]/message' = N[others]/message
      }
    ] {}

    default state Unassigned {
      trans RootAssign {
        when {
          this in root
        }
        do {
          level' = nodeLevel/first
          parent' = this
        }
        goto Assigned
      }

      trans NodeAssign {
        when {
          some message
        }
        do {
          level' = nodeLevel/next[Node.message]
          parent' = message.Level
        }
        goto Assigned
      }
    }

    state Assigned {
      trans sendMessage {
        when {
          some n: Node | no N[n]/level
        }
        do {
          (sendMessage)
        }
      }
    }

    init {
      no level
      no parent
      no message
    }
  }
}
```

Appendix C

Heating System Model

```
open util/ring[Node] as node
open util/ordering[Temp] as temp

sig Temp{}
abstract sig ValvePos {}
one sig OPEN, HALF, CLOSED extends ValvePos {}

conc state HeatingSystem {
  env event Reset {}
  env event TurnOn {}
  env event furnaceFault {}
  env event userReset {}
  env event heatSwitchOn {}

  event activate {}
  event deactivate {}
  event furnaceRunning {}
  event furnaceNotRunning {}
  event furnaceReset {}

  default state Functioning {
    conc state Furnace {
      default state Furnace_Normal {
        default state Furnace_Off {
          trans T1 {
```

```
          on activate
          goto Furnace_Activating
        }
      }
      state Furnace_Activating {
        trans T2 {
          on deactivate
          goto Furnace_Off
        }
        trans T3 {
          send furnaceRunning
          goto Furnace_Running
        }
      }
      state Furnace_Running {
        trans T4 {
          on deactivate
          goto Furnace_Off
        }
        trans T5 {
          on furnaceFault
          goto HeatingSystem/ERROR
        }
      }
    }
  }

  conc state Controller {
    controllerOn: one Bool
    condition heatRequested [some r: Identifier |
      Functioning/Room[r]/requestHeat = True] {}
    condition noHeatRequested [no r: Identifier |
      Functioning/Room[r]/requestHeat = True] {}

    default state Off {
      trans T8 {
        on heatSwitchOn
        send furnaceReset
        do controllerOn' = True
        goto On
      }
    }

    state On {
      default state Idle {
        trans T9 {
          when heatRequested
          send activate
          goto Heater_Active
        }
      }
      state Heater_Active {
        trans T10 {
          when noHeatRequested
          send deactivate
          goto Idle
        }
        trans T11 {
          on furnaceFault
          do controllerOn' = False
          goto HeatingSystem/ERROR
        }
      }
    }

    init {
      controllerOn = False
    }
  }

  conc state Room [Identifier] {
    actualTemp: one Temp
    desiredTemp: one Temp
    valvePos: one ValvePos
    requestHeat: one Bool

    env event waitedForWarmth {}
    env event waitedForCool {}

    condition tooCold [lt[actualTemp, desiredTemp]] {}
    condition tooHot [gt[actualTemp, desiredTemp]] {}
    condition vOpen [valvePos = OPEN] {}
    condition vClosed [valvePos = CLOSED] {}
    condition controllerOn [Functioning/Controller/
  controllerOn = True] {}
    action rH [requestHeat' = True] {}
    action cancelrH [requestHeat' = False] {}

    default state No_Heat_Request {
      default state Idle_No_Heat {
        trans T12 {
          when tooCold
          goto Wait_For_Heat
        }
        trans coolRoom {
          when !tooCold
          do actualTemp' = temp/prev[actualTemp]
        }
      }
      state Wait_For_Heat {
        trans T13 {
          when !(tooCold)
          goto Idle_No_Heat
        }
        trans T14 {
          on waitedForWarmth
          when valvePos = CLOSED
          do valvePos' = OPEN
        }
        trans T15 {
          when vOpen and controllerOn
          do rH
          goto Heat_Requested
        }
      }
    }

    state Heat_Requested {
      default state Idle_Heating {
        trans T15 {
          when tooHot
          do valvePos' = CLOSED
          goto Wait_For_Cool
        }
        trans heatRoom {
          when !(tooHot)
          do actualTemp' = temp/next[actualTemp]
        }
      }

      state Wait_For_Cool {
        trans T16 {
          when !(tooHot)
          goto Idle_Heating
        }
        trans T17 {
          on waitedForCool
          do valvePos' = CLOSED
        }
        trans T18 {
          on waitedForCool
          when vClosed
          do {
            cancelrH
            actualTemp' = desiredTemp
          }
          goto No_Heat_Request
        }
      }
    }

    init {
      requestHeat = False
      valvePos = CLOSED
    }
  }
}

state ERROR {
  trans T19 {
      on heatSwitchOn
      goto Functioning
    }
  }
}
```