Verifying Distributed Controllers with Local Invariants

Yiqun Wang, Shengwei An, Xiaoxing Ma, Chun Cao, and Chang Xu
State Key Laboratory for Novel Software Technology at Nanjing University
Institute of Computer Software, Deptartment of Computer Science and Technology, Nanjing University
Nanjing 210023, Jiangsu, China
{wyqqrince, njuasw, xiaoxing.ma, caochun}@gmail.com, changxu@nju.edu.cn

***Abstract*—Controllers restrict systems to behave only in good manners. Different from controlling monolithic systems where controllers can be automatically synthesized from specifications, controlling distributed systems often has to use *distributed* controllers that are *manually* programmed. To ensure their correctness, manually programmed controllers themselves need to be formally verified. This task can be challenging due to the complexity caused by the autonomy and asynchrony of distributed controllers. The limited scalability of existing model checkers also exacerbates the problem. In this paper we explore the modeling and verification of distributed controllers using Alloy. Besides resorting to the *Small Scopes Hypothesis* of the Alloy methodology, we also leverage local invariant based modular verification techniques for better scalability. A local invariant characterizes a logical relationship between a local sub-system and its neighbors and abstracts away the concrete interactions. These concrete interactions would otherwise explode the system state space during verification. The approach is first illustrated with the well-understood Two-Phase Commit protocol, and then is applied to the verification of several dynamic software update protocols, which gives an initial evidence of its effectiveness.**

***Keywords*—Distributed Controllers, Alloy, Modular Verification, Dynamic Software Update**

I. Introduction

Software systems are increasingly ubiquitous in everyday life. It is essential to guarantee that software should behave in a correct and reliable manner as life might depend on it. Controllers provide a way to ensure the systems behave themselves. A (discrete event) controller [1], [2] restricts the occurrence of actions it controls based on its observation of the system's actions that have occurred.

Controllers of monolithic systems can be synthesized automatically from high-level specifications [3], [4]. However, it can be challenging to control distributed systems [5]. Using centralized controller is not an option in many cases, because it would require consistent global snapshots of the system under control, and taking global snapshots of distributed systems could be very expensive [6]. Therefore, we prefer distributed controllers. Each controller is deployed locally with a sub-system. A controller observes actions of its sub-system and communicates with other controllers if necessary. However, we can hardly synthesize distributed controllers automatically because of their complexity. Instead, they are created manually.

Verifying manually distributed controllers is highly motivated, because faulty controllers may lead the system to behave worse or in unexpected ways even if the controlled systems are free from errors, which will mislead developers to check faultless parts. As a representative of traditional verification techniques, model checking is fully automatic and is capable of producing a counterexample. However, it can handle very limited instances of concurrent systems, mostly because of the very large number of possible states and of possible interleavings of executions. The state space of a concurrent system with distributed controllers booms even faster, since distributed controllers ensure that the system meets global properties, which requires cooperation and communication of distributed controllers. It cannot be decomposed into subproblems where each part of the global properties is verified locally and separately.

To address that, we give an exercise of verifying distributed controllers with Alloy [7]. Alloy is a lightweight specification modeling tool and it has been successfully applied to a wide range of application domains [8], [9], [10], [11]. With Alloy, the scopes of verification could be restricted since the designers of Alloy justify the decision work within limited scopes through an appeal to *Small Scope Hypothesis* [12]. While bounded verification with a certain scope is typically faster than exhaustive verification, it can still be a daunting task for distributed systems with unlimited behaviors. Hence we intend to integrate modular verification [13], [14], [15] with Alloy to support better scalability. In our approach, we first devise some local invariants. A local invariant characterizes a logical relationship between a local sub-system and its neighbors and abstracts away the concrete interactions. Then we leverage them in the specification to make the verification process more efficient.

We use controllers in the Two-Phase Commit (2PC) protocol [16] and several dynamic software update (DSU) protocols to demonstrate our approach. The 2PC protocol is a classic transaction commit protocol which helps to ensure the consistent termination of a number of transactions, whether to commit or abort. The transaction should be committed only if all parts of system are willing to commit it and should be aborted if any part of system chooses to abort it. The 2PC protocol is widely utilized since it can achieve its goal even in many cases of system failures involving process failures and communication failures. DSU, unlike offline update, updates the running systems on the fly. DSU is not trivial because it is difficult to guarantee the correctness, since it might lead to misbehaviors that never present in either versions. To ensure the correctness of DSU, target component should reach a safe state. In the past decades, many algorithms and protocols [17], [18], [19] have defined what a safe state for DSU is.

This paper has two intended contributions. First, we propose a carefully designed modularization technique to improve the scalability of the verification of distributed controllers in Alloy. We devise some local invariants which characterize logical relationships between each local sub-system and its neighbors, and leverage them for efficient verification. Second, our case study provides a formal verification of the correctness of several DSU protocols [17], [18], [19], whose original proofs are semi-formal.

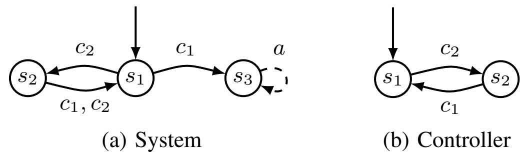
The rest of paper is organized as follows. Section II introduces backgrounds of controllers and Alloy. Section III gives the overview of our approach and uses 2PC protocol to demonstrate our approach. Section IV applies our approach to DSU protocols. Section V discusses related work and section VI concludes the paper.

II. Background

A. Controllers

Consider a system N represented by all its possible behaviors. This system can be viewed as a *plant* to be controlled [20]. The plant's behaviors satisfying some properties are defined as good ones. A controller $\mathcal{M}$ is another system restricting the *plant* to behave only in good ways. Following [21], an action of the *plant* is monitored/controllable if such action is uncontrollable/controllable by the controller. Controller restricts the occurrence of controllable actions based on its observation of the *plant*'s actions.

As a simple example, consider the system in Fig. 1a. A node denotes a state of system and an edge denotes an action. Suppose we want to use a controller to ensure that system cannot have action a but must have action c_1. The controller in Fig. 1b is a solution. The dashed action a cannot be controlled by the controller while others can. It forbids the appearance of action a by restricting transition from s_1 to s_3, since if the system has reached state s_3, action a is about to happen. In addition, the controller prohibits the transition on action c_2 from s_2 to s_1 to guarantee the appearance of action c_1.



(a) System (b) Controller

Fig. 1: A controller example

B. The Alloy Specification Language

Alloy is a lightweight declarative relational modeling tool. It consists of the *Alloy language* and the *Alloy Analyzer*. The *Alloy language* is simple but expressive, and its syntax, based on first-order logic, is designed to make it easy to build models incrementally. The *Alloy Analyzer* is a solver that takes the specification of a model and locates satisfiable instances by converting it into SAT. A simple object model of binary tree is presented as follows.

```
sig Key {}
sig Tree { root : one Node}
sig Node {
  left, right : lone Node,
  value : Key,
}

fun parent(): Node->Node {~(left + right)}

fact {
  all n : Node | n.right != n and n.left != n
  all n : Node | some n.(left + right) implies n.
      left != n.right
  all n : Node | (some t : Tree | t.root = n and no
      n.parent) or one n.parent
}

pred NoDisjointValue {
  no disj n1, n2 : Node | n1.value = n2.value
}

assert Acyclic {no n : Node | n in n.^parent}

run NoDisjointValue for 5
check Acyclic for 10
```

Fig. 2: A simple model of binary tree in Alloy

The *Alloy language* uses *signature* to indicate the existence of disjoint sets or atoms. This object model consists of three signatures, **Key**, **Tree** and **Node**. The **Key** signature contains no field and is used here to represent a string value. The **Tree** signature contains a field **root** which is a binary relation that maps every tree to exactly one node. The **Node** signature contains three binary relations. The **left** and **right** relations map nodes to their children and **value** maps nodes to keys. The qualifier *lone* indicates that every node is related to zero or one node with the **left** and **right** relations.

The model also includes a *function*, a *fact*, a *predicate* and an *assertion*. In general, an Alloy function denotes a relation between its arguments and the result. The **parent** function returns a binary relation maps nodes to nodes. We can use *n.parent* to represent the parent of *n*. The constraints in the fact are always assumed to hold. In above model, we assume that every node cannot have a **left** or **right** relation to itself and that its **left** and **right** are disjoint and that every node has one parent except the root node. The constraints in the predicate are assumed to hold when invoked. For instance, the binary tree model might require that the value of every node is unique. The constraints in assertion are the intended properties in the model, *i.e.*, properties to be checked.

A key advantage of using Alloy as modeling language is that object models can be analyzed fully automatically. Although first-order logic is undecidable, it is possible to analyze Alloy models by restricting the search space to a certain finite scope. The analysis can be used either to explore the model by generating sample instances, or to check properties by trying to generate counterexamples. The *Alloy Analyzer* achieves this by converting the model into a propositional CNF formula, and exhaustively searching for instances within user-defined scope. Running the last two commands in above model, we can get instances where the value of every node is unique and check the property that no cycles exists in a binary tree.

The core of the *Alloy Analyzer* is implemented as a model-finder. In order to ensure the model-finding problem is decidable, the *Alloy Analyzer* performs model-finding over restricted scopes by resorting to *Small Scope Hypothesis* [12], which shows that a high proportion of bugs and errors can be found by searching all instances within some small scopes. Hence exhaustive searching and checking within a small scope are worth pursuing.

III. Our Verification Approach

A. Approach Overview

The general architecture of our approach consists of two parts, controlled system and controllers. An example is shown in Fig. 3. Each rectangle denotes a local controlled system, each circle denotes a sequence of its actions, controllers are denoted by arcs. Each local controlled system communicates with others by passing messages, denoted by little triangles.

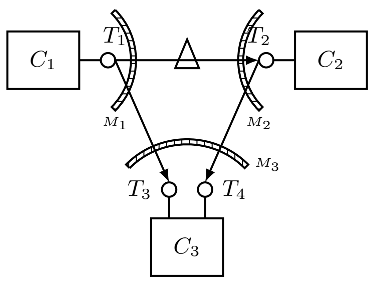


Fig. 3: A distributed system example

Consider controlled system N consists of a set of components $\{C_1, C_2, \ldots, C_n\}$. Each component can be regarded as a labelled transition system (LTS) (S, s_0, L, Δ), where

- S is a set of the component's states.
- $s_0 \in S$ is the initial state.
- L is a set of action labels. Local action means local computation, remote action indicates exchanging data through messages.
- Δ is a set of transition relations $(S \times L \times S)$. The fact that $(s_i \times l_j \times s_k) \in \Delta$ is written as $s_i \xrightarrow{l_j} s_k$. Specially, $s_i \xrightarrow{l_j} s_{i+1} \xrightarrow{l_h} s_{i+2}$ can be simpled as $s_i \xrightarrow{l_j l_h} s_{i+2}$.

It is assumed that LTS is *action-deterministic*. This entails that for any state $s \in S$ and any action $l \in L$, s has most one outgoing transition which with action l. Formally, $s \xrightarrow{l} s'$ and $s \xrightarrow{l} s''$ implies $s' = s''$. The execution of a LTS can be represented by a continuous sequence of actions (the initial state can be ignored). We can use $C_1 \parallel C_2 \parallel \cdots \parallel C_n$ to denote N.

In our architecture, controlled system is an asynchronous concurrent system. The assumed communication pattern is such that each pair of communicating components is connected by two simple channels. The ordering among the messages is nondeterministic which results in messages arriving in unpredictable order. As a consequence of this nondeterminism, controlled system may show different behaviors for the same configuration and environment. An execution of controlled system can be considered as an interleaving of the collection of each component's executions. For example, in Fig. 3, each component's executions might be

$$\begin{aligned} C_1 &: \langle S_{m_1}, LC, S_{m_2}, R_{m_6}, R_{m_4} \rangle \\ C_2 &: \langle R_{m_1}, LC, S_{m_3}, R_{m_5}, LC, S_{m_6} \rangle \\ C_3 &: \langle R_{m_2}, R_{m_3}, LC, S_{m_4}, S_{m_5} \rangle \end{aligned}$$

LC stands for local calculation. S_{m_i} stands for the action of sending a message m_i. R_{m_i} stands for the action of receiving a message m_i. We assume there exists four types of messages:

- *create* message: a message calling remote actions
- *finish* message: a message carrying the consequence of remote actions
- *data* message: a message denoting exchange of data
- *ack* message: a message of acknowledgment

In the execution of controlled system, action S_{m_1} should happen before R_{m_1} since message m_1 cannot be received before it has been sent. Moreover, S_{m_1} should happen before S_{m_2}, as S_{m_1} is in front of S_{m_2} in the execution of C_1.

To ensure a property φ, we define a set of controllers $\mathcal{M}_\varphi = \{M_1, M_2, \ldots, M_n\}$. Intuitively, a set of correct controllers work together with their respective components to ensure that no traces will violate φ, denoted by $\|_{i=1}^{n}(C_i \parallel M_i) \vDash \varphi$. For simplicity, we use $N \parallel \mathcal{M}_\varphi$ to denote $\|_{i=1}^{n}(C_i \parallel M_i)$.

Definition 1. Given a property φ and a system N, a set of controllers $\mathcal{M}_\varphi$ is *correct* if we have $N \parallel \mathcal{M}_\varphi \vDash \varphi$.

The scope of verifying controllers might be unbounded as the system's size can be arbitrarily large. However, with the concept of *Small Scope Hypothesis* in Alloy, the scope of verification could be restricted by parameterizing the number of components, controllers and messages.

B. Modeling and Verifying Two Phase Commit Protocol

We use the well-understood 2PC protocol [16] to illustrate our approach. In the 2PC protocol, a transaction is performed by a collection of processes called *resource managers* (RMs), each executing on a different node. The fundamental requirement is that all RMs must eventually agree on whether the transaction is committed or aborted. The goal of the 2PC protocol is to reach *committed* or *aborted* state for all RMs.

The 2PC protocol uses a *transaction manager* (TM) process to coordinate the decision-making procedure. TM has following state: *init*, *preparing*, *committed*, and *aborted*. TM is in *init* and all RMs are in *working* state at the beginning. The 2PC protocol starts when an RM enters *prepared* state and sends a *Prepared* message to TM. Upon receiving the message, TM enters *preparing* state and sends *Prepare* messages to other RMs. Upon receiving a *Prepare* message, an RM will enter *prepared* state and reply a *Prepared* message. When TM has received *Prepared* messages from every RMs, it enters *committed* state and sends *Commit* messages to all RMs. Each RM enters *committed* state upon receiving *Commit* message. An RM can spontaneously enter *aborted* state and send an *Abort* message to TM if it is in *working* state, and TM can spontaneously enter *aborted* state unless it is in *committed* state. When TM aborts or it receives an *Abort* message from RM, it sends *Abort* messages to all RMs. Upon receiving of such a message, an RM enters *aborted* state. The state transition diagram for RM is in Fig. 4 (*sp* stands for sending *Prepared* message, *sa* stands for sending *Abort* message, *rc* stands for receiving *Commit* message, *ra* stands for *Abort* message).

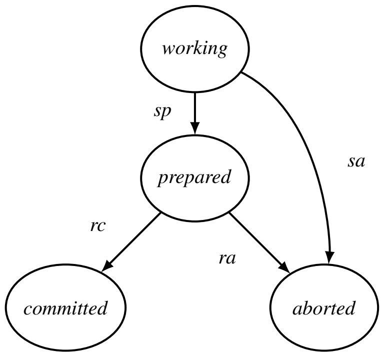


Fig. 4: State transition diagram of RM

The details of the 2PC protocol is implemented in a set of distributed controllers. To precisely model distributed systems, time is a significant and indispensable notion. However, Alloy has no built-in notion of time, so we leverage tick-based modeling [9] to represent the order of actions. This total order can be defined by applying ordering functions from the Alloy library to signature **Tick**. It is worth mentioning that the total order neither represents a consistent global state of a distributed system nor synchronizes distributed controllers. Instead, it represents the execution of a distributed system by interleaving each component's execution (see the constraints in Fig. 8 and Fig. 15).

```
open util/ordering[Tick]
sig Tick{}
```

Fig. 5: Signature of Tick

We assume that both RM and TM are elements of signature **Process** (see Fig. 6), and the controller of a **Process** will record a state. For simplicity, TM has three states: **init**, **committed** and **aborted**; RM has four states: **working**, **prepared**, **committed** and **aborted**. What's more, the controller of TM will also manage a set called **tmPrepared** to record the RMs whose states are **prepared**.

Messages can be lost or duplicated, but not corrupted. Besides, the delay of message is nondeterministic. To simplify model, we only consider three types of messages: **Prepared**, **Commit** and **Abort**. We eliminate the *Prepare* message sent by an RM and assume that all RMs can spontaneously issue *Prepared* messages. We also ignore the *Abort* messages sent by an RM when it decides to abort. Such a message would cause the TM to abort the transaction, represented by the TM spontaneously deciding to abort[1].

```
abstract sig PState {}
one sig init, working, prepared, committed, aborted
    extends PState{}
abstract sig Process { state : Tick -> PState }
one sig TM extends Process {
  tmPrepared : Tick -> set RM
}
sig RM extends Process{}
abstract sig MsgType{}
one sig Prepared, Commit, Abort extends MsgType{}
sig Message {
  type : MsgType,
  sender : Process,
  receiver : Process,
  sendTime : Tick,
  receiveTime : lone Tick
} {
  sender != receiver
  receiveTime in Tick implies sendTime in
      receiveTime.prevs
}
```

Fig. 6: Snapshot of 2PC model

We consider the execution of system consists of the following event:

- TM has received **Prepared** messages from all RMs and sends **Commit** messages to all RMs.
- TM spontaneously decides to abort transaction and sends **Abort** messages to all RMs.
- TM receives a **Prepared** message from a RM.
- RM spontaneously sends a **Prepared** message to TM.
- RM spontaneously decides to abort.
- RM receives a **Commit** message from TM.
- RM receives a **Abort** message from TM.

For example, if an RM spontaneously decides to abort, its state should transit from **working** to **aborted**. The state of other RMs and TM should remain unchanged. Since we ignore the *Abort* message sent by an RM, no message is being sent right now. The **tmPrepared** set is not changed as well.

[1]For more details: https://lab.artemisprojects.org/wyq/AlloyCode

```
pred RMChooseToAbort(rm : RM, t : Tick) {
  rm.state[t.prev] = working
  rm.state[t] = aborted

  all rm' : RM - rm | rm'.state[t] = rm'.state[t.
      prev]
  TM.state[t] = TM.state[t.prev]
  TM.tmPrepared[t] = TM.tmPrepared[t.prev]
  no m : Message | m.sendTime = t
}
```

Fig. 7: Predicate of a RM chooses to abort transaction

Then an execution of the 2PC protocol can be specified as:

```
pred running2PC {
  Init
  all t : Tick - first | TMCommit[t] or TMAbort[t]
      or {some rm : RM | TMRcvPrepared[rm, t] or
      RMPrepare[rm, t] or RMRcvCommitMsg[rm, t] or
      RMChooseToAbort[rm, t] or RMRcvAbortMsg[rm, t
      ]}
}
```

Fig. 8: Predicate of running the 2PC protocol

The controllers of the 2PC protocol chooses to commit the transaction if the RM is in **committed** or to abort if its state is **aborted**. In order to ensure the consistent termination of a transaction, two *safety* properties (φ of the 2PC protocol) should be guaranteed:

- *Stability*: Once an RM has entered *committed* or *aborted* state, it remains in that state forever.
- *Consistency*: It is impossible for one RM to be in *committed* state and another to be in *aborted* state.

In Fig. 9, we assume that the states of controllers will not violate *stability* and *consistency*:

```
pred consistency {
  all t : Tick | no disj r1, r2 : RM | r1.state[t] =
       aborted and r2.state[t] = committed
}

pred stability {
  all t : Tick, rm : RM | rm.state[t] = committed
      implies rm.state[t.nexts + t] = committed
  all t : Tick, rm : RM | rm.state[t] = aborted
      implies rm.state[t.nexts + t] = aborted
}

assert safetyOf2PC {
  running2PC implies (consistency and stability)
}
```

Fig. 9: Safety property of 2PC protocol

We conducted our experiments on an eight-core machine with Intel Core i7 CPU @3.40GHz and 4GB RAM, running Windows 8. Table I shows that controllers of the 2PC protocol satisfy *consistency* and *stability* under different bounded scopes. The scope of these verification is not the number of RMs but the number of all signatures (**Process**, **Tick** and **Message**). The experimental result shows that the controller implementation of the 2PC protocol is correct. Therefore our approach to verifying distributed controllers works.

TABLE I: Result of 2PC controllers

Scope	#Clauses	Time(s)	Counterexample
8	49911	8.188	No
9	70717	32.690	No
10	95265	47.598	No

C. Local Invariant Based Modular verification

However, the verification might be overwhelming even in some reasonable scopes. Although we cannot decompose the verification of global properties into subproblems of smaller ones, we can devise some local properties and use them to help verifying global ones. The local properties usually describe the execution in a more concise way by formalizing the logical relationships between sub-systems. They can be considered as an assumption when verifying other parts of the system, which presents the basic idea of modular verification [13], [14], [15].

In LTS, we let $Act(s) = \{l \in L \mid \exists s' \in S \cdot s \xrightarrow{l} s'\}$ denotes the set of actions that are *enabled* in state s. For any state s and $l \in Act(s)$, let $l(s)$ denotes the unique l-successor of s, *i.e.*, $s \xrightarrow{l} l(s)$. More generally, if $s \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} \dots \xrightarrow{l_n} s_n$, we say that $s \xrightarrow{L_i} L_i(s)$ where $L_i = l_1 l_2 \dots l_{i-1} l_i$. We give a definition of independent actions.

Definition 2. Let a LTS (S, s_0, L, Δ) be an *action-deterministic* transition system with $l_1, l_2 \in L$, $l_1 \neq l_2$.

- l_1 and l_2 are *independent* if for any $s \in S$ with $l_1, l_2 \in Act(s)$:
  - $l_2 \in Act(l_1(s))$
  - $l_1 \in Act(l_2(s))$
  - $l_1(l_2(s)) = l_2(l_1(s))$
- l_1, l_2 are *dependent* if l_1 and l_2 are not independent.

According to Def. 2, it is obvious that for each pair of independent actions, the final state has nothing to do with the order of their occurrences. For example in Fig. 10, s_1 can reach s_2' by executing action sequence ll' or $l'l$.

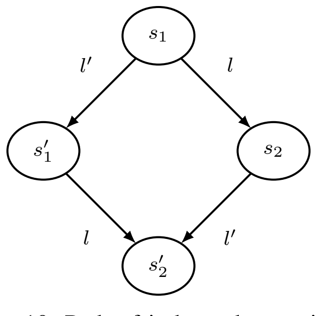

$$\begin{tikzcd} & s_1 \arrow[dl, "l'"'] \arrow[dr, "l"] & \\ s_1' \arrow[dr, "l"'] & & s_2 \arrow[dl, "l'"] \\ & s_2' & \end{tikzcd}$$

Fig. 10: Path of independent action

Given a LTS where each pair of actions are independent, we split the state space into two parts according to a specific (resp. set of) action $l \in L$ (resp. $l \subseteq L$). One part denoted by $H(l)$ stands for l has already happened, and the other part denoted by $N(l)$ stands for l has not happened. We intend to find some properties, denoted by P_l, that lie in either $H(l)$ or $N(l)$, *i.e.*, they are "frozen" unless l is occurring. Algorithm 1 shows how to get $H(l)$, $N(l)$ and P_l. Our intention of Algorithm 1 is explained as follows.

Algorithm 1 Get $H(l)$, $N(l)$ and P_l of l

```
Input: The action $l$, a LTS = $\{S, s_0, L, \Delta\}$
Output: $H(l)$, $N(l)$ and $P_l$
$P_l = \emptyset$;
$N(l) = s_0, H(l) = l(s_0)$;
Obtain $P_l$ when $s_0 \xrightarrow{l} l(s_0)$ happen;
$S = (L - l)(s_0)$;
while $N(l)$ changed do
    $S' = S, S = \emptyset$;
    for $s \in S'$ do
        for $l' \in Act(s)$ do
            $S = S \cup l'(s)$;
            $N(l) = N(l) \cup l'(s)$;
            $H(l) = H(l) \cup l(l'(s))$;
            for $p \in P_l$ do
                if $p$ remains when $l'(s) \xrightarrow{l} l(l'(s)$ then
                    $P_l = P_l - p$;
                end if
            end for
        end for
    end for
end while
```

First, P_l is initialized when l occurs at state s_0 by observing changes of memory, disk or other user-defined variables. Algorithm 1 removes an element $p \in P_l$ if it remains when another transition on l is occurring, as P_l cannot belong to both $N(l)$ and $H(l)$. Finally, the rest of the elements in P_l is what we want. P_l can describe some characteristics in LTS referring to action l. Suppose we gather sufficient number of actions or select several distinguishable ones among them, the execution of LTS could be expressed more concisely using P_l. Thus we might achieve an efficient verification using them as an environment assumption in the model.

As it is hard to guarantee the independence of actions in a distributed system, we separate it into several blocks where each pair of actions is independent. Intuitively, sending and receiving of messages should be separated into different blocks since a message cannot be received before it is sent.

D. Local Invariants in the 2PC Protocol

As we discussed above, in the 2PC protocol, TM sends **Commit** messages only if it has received **Prepared** messages from all RMs, otherwise it sends **Abort** messages. We see that the action of TM sending messages is dependent to the action of RM sending messages. In order to apply modular verification, we split the 2PC protocol into two parts. In one part, only TM is capable of sending messages, and in the other part, only RM is. These two parts just correspond with the two phases of the 2PC protocol:

- *Phase1*: TM attempts to prepare all the RMs to take the necessary steps for either committing or aborting transactions.
- *Phase2*: Based on the results, TM decides whether to commit or abort the transaction, and notifies the result to all RMs. Then each RM follows with needed actions (commit or abort) with its location resources and its respective portions in output.

In *Phase1*, only RM is allowed to send messages. Upon sending messages, the state of RM will become **prepared** or **aborted**; all messages are received by TM, it will add corresponding RM to set **tmPrepared** upon receiving each **Prepared** messages. In other words, the transition of state of RM is only related to its communication with TM. Similarly, in *Phase2*, TM will broadcast messages to all RMs. The state transition of RM is only allowed when receiving such message. If we choose l are actions of RM's sending messages in *Phase1* and actions of RM's receiving messages in *Phase2*, constraints of P_l represent RM's state transition.

```
pred StateTransition (rm : RM) {
  let m_r = {m : Message | m.receiver = rm} |
    let t2 = m_r.receiveTime | {
      m_r.type = Abort implies rm.state[t2] =
          aborted
      m_r.type = Commit implies rm.state[t2] =
          committed
      {no m : Message | m.sender = rm} implies {
        all t : Tick - first - t2 | rm.state[t.prev]
             = working implies rm.state[t] in (
            working + aborted) else rm.state[t] = rm
            .state[t.prev]
      } else {
        let t1 = {m : Message | m.sender = rm}.
            sendTime | {
          all t : Tick - first - t2 - t1 | rm.state[
              t] = rm.state[t.prev]
          rm.state[t] = prepared
          rm.state[t.prev] = working
        }
      }
    }
}
```

Fig. 11: Predicate of state transition in the 2PC protocol

To be specific (see Fig. 11), in *Phase1*, its state is **working** until it decides to commit (state becomes **prepared**) or abort (state becomes **aborted**); in *Phase2*, its state will transit to **committed** when receiving **Commit** message or to **aborted** if it is **Abort** message. The rules of state transitions will be specified as the assumption of controllers' behaviors. Similarly, we assume that the new specification will not violate *consistency* and *stability* as well (see Fig. 12), the result is shown in Table II.

TABLE II: Result of verification in local invariants

Scope	#Clauses	Time(s)	Counterexample
8	24026	4.141	No
9	32366	6.782	No
10	40854	11.189	No

```
assert SafetyOf2PCinLocalInvariants  {
  (Init and StateTransition) implies (consistency
      and stability)
}
```

Fig. 12: Assertion of verification in local invariants

In Table II, it shows that after integrating modular verification, controllers of the 2PC protocol will also guarantee *safety* properties. We extend the comparison to different scopes, the result is shown in Fig. 13.

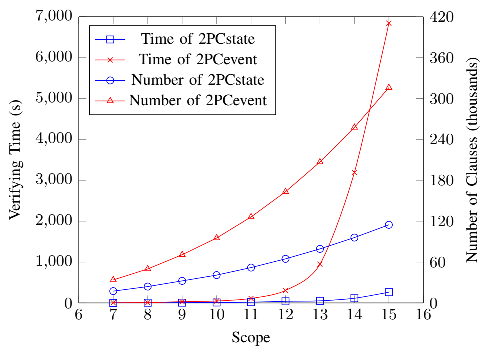


Fig. 13: Comparison of 2PCevent and 2PCstate

In Fig. 13, 2PCevent represents the non-modular approach while 2PCstate represents the modular one. It shows that 2PCstate is more efficient than 2PCevent in both the size of specification and the time of verification. On increasing the number of scope, the effect becomes more phenomenal.

IV. Modeling and verifying DSU protocols

In this section, we use controllers in DSU protocols to demonstrate our approach. We first talk about the reason to choose them as subjects. After we introduce these protocols, we show the modeling details in Alloy. We then give a result of verification and some improvements by using local invariants.

The reason why we choose DSU protocols as our subjects is as follows. First, the problem of DSU (of CBDS) contains typical characteristics of distributed systems. The basic artifact is a component, it holds services that can be accessed remotely. Components may also lean on services provided by other components. In addition to remote communication, components also need local calculation. Second, DSU protocols can be considered as a representative problem of distributed controllers. The target component is allowed to update when it meets certain conditions [17], [18], [19]. Third, the details of these protocols are not trivial. The consistency of them depends on the cooperation of each local part, any mistakes can result in an unexpected consequence. A precise implementation of controllers could help to discover the essential cause of an inconsistency. Last but not least, original proofs of these protocols are semi-formal, our approach could provide a formal verification of their correctness.

A. *Dynamic Software Update Protocol*

Normally, the update of running program consists of shutting down the system, installing new version and restarting the system. However, in some specific domains, such as financial transaction processing and transport controlling, software should be online 24/7 and non-stopping service is mandatory. It is therefore necessary to update the existing systems at runtime. When compared to the offline update, DSU is more difficult to guarantee the correctness, because it might lead to misbehaviors that would never present in old or new version of the program. To ensure the correctness, target component must satisfy a sufficient condition when updating. In the past decades, many algorithms and protocols [17], [18], [19] have defined what a sufficient condition for DSU is.

Kramer *et al.* [17] proposed a criterion called *quiescence* as a sufficient condition for a node to be safely manipulated during dynamic reconfigurations. Vandewoude *et al.* [18] proposed a definition of *tranquillity* to address the high disruption (the interruption of system's services) of *quiescence* protocol. However, the limitation of *tranquillity* protocol is that it mainly focuses on local consistency [19]. Ma *et al.* [19] used the notion of *version consistency* as a sufficient criterion for the safety of DSU. They also introduced a management framework (VC) on dynamic edges that are labelled as either *future* or *past*. A *future* (resp. *past*) edge $C \to C'$ implies that C might request (resp. have already requested) a service provided by C' in the future (resp. past). They claimed that a component C is said to be *free* iff. there is not a pair of future/past edges entering C. They proved that a dynamic update of a component C satisfies *version consistency* when C is *free*.

B. *Modeling in Alloy*

For different DSU protocols, model of controller implements details of protocols, and model of controlled system describes its execution. We also leverage **Tick** in the model of DSU protocols.

Version is the fundamental concept in DSU protocols. To put it simply, we define two versions which represent either old or new version. The version of a component will not change unless it is updated.

```
abstract sig Version{}
one sig Ver_0, Ver_1 extends Version{}
```

Fig. 14: The signature of Version

A component represents a local controlled system. The service of a component is represented by transactions. If a transaction T_1 of component A calls another transaction T_2 of component B, we say that A is static independent to B. A transaction can only be hosted by one component but a component can provide an arbitrary number of transactions (see Fig. 15). Each transaction has three states: **OFFLINE**,**ONLINE** and **END**. We assume:

- All transactions are **OFFLINE** at the very beginning.

```
sig Component{
  transaction: set Transaction,
  version: Tick → Version,
  staticEdge: set Component,
}

abstract sig TxState{}
one sig OFFLINE, ONLINE, END extends TxState{}
sig Transaction{
  hostComponent: Component,
  subTransaction: set Transaction,
  hostVersion: Version,
  state: Tick → TxState
}

abstract sig MsgType{}
one sig INIT, FINISH, ACK_I, ACK_F extends MsgType{}
sig Message {
  from: Component,
  to: Component,
  sender: Transaction,
  receiver: Transaction,
  type: MsgType,
  sendTime: Tick,
  receiveTime: Tick
} {
  from!=to
  sender!=receiver
  from=sender.hostComponent
  to=receiver.hostComponent
  sendTime in receiveTime.prevs
}

pred ControlledSystem {
  running
  all t : range[Root.beginTime.next, Root.endTime.
      prev] |
    Update.updateTime = t or some msg : Message |
        msg.sendTime = t or msg.receiveTime = t
}
```

Fig. 15: A snapshot of controlled system

- A transaction T can be initiated by an outside client or by its *parent-transaction* T'. T is called *root-transaction* in the former case and a *sub-transaction* (of T') in the latter case. After initiation, its state becomes **ONLINE**.
- The *hostVersion* of a transaction represents the version of its hosted component when it is initiated.
- Whenever a transaction is **ONLINE**, it can initiate its *sub-transaction* whose status is **OFFLINE**.
- A transaction is not finished until all its *sub-transactions* are finished.
- When a transaction is finished, its state becomes **END**.
- A process of controlled system begins with the initiation of *root-transaction* and ends when *root-transaction* is accomplished.

The transactions communicate by passing messages. To simplify model, a transaction cannot send messages to another one if they share the same component, otherwise these two transactions can be merged into a bigger one and this internal message can be ignored. In order to indicate the delay in processing messages, we let a message be sent at some time and received by others at any later time. Besides, the messages is not lost or corrupted.

Since DSU protocols are not interested in the business logics, we only consider four types of message: **INIT** and **FINISH**, representing initiation or accomplishment of a transaction; **ACK_I** and **ACK_F**, representing the reply. The reason why we involve **ACK** message is that without them, the asynchrony of distributed systems will cause incorrect state of controllers, which might result in an unexpected fault. We use the example in Fig. 3 to explain our intention. T_1 is C_1's transaction and *root-transaction*, T_2 is C_2's transaction, T_3 and T_4 are C_3's transactions. Suppose we are using VC protocol to update C_3. At the beginning, when T_1 is **ONLINE**, it will initiate T_2 and T_3, after that C_1 knows it will not use C_2 and C_3, so C_1 removes the *future* edges to C_2 and C_3. If T_2 receives the message earlier, it becomes **ONLINE** and initiates T_4. T_4 receives the message and begins with old version of C_3. After the termination of T_4 and T_2, the **INIT** message from C_1 to C_3 has not been received yet. Now we can update C_3 component to new version since it is *free* now. However, after update, T_3 will eventually receive the **INIT** message and start with new version of C_3. Under the circumstances, T_4 and T_3 violate consistency. This unexpected result comes from wrong implementation of controllers, the management of dynamic edges does not correspond with the rules of VC protocol. To avoid this, **ACK** messages should be involved in specifications.

The signature of **update** consists of a target component, the update time and the new version of the target component, which is different from the original version.

```
one sig Update{
  component: Component,
  updateTime: Tick,
  newVersion: Version
} {newVersion != component.version[first]}
```

Fig. 16: A snapshot of DSU controllers

Quiescence protocol [17] allows dynamic update only when the target component is *quiescent*. A component is *quiescent* iff. all of its dependent components is *passive*. In our model, a component is *passive* iff. none of its transaction is **ONLINE** right now.

Tranquillity [18] and VC [19] protocols are similar. Tranquillity protocol only leans on whether a component has been used or will be used by one adjacent component, but VC protocol focuses on all dependent components. Combining the management algorithm [19] in VC protocols with our controlled system, each controller will be managing four edges set: *future-in*, *future-out*, *past-in* and *past-out*. *Future-in* of component C records the components which have a future edge pointing to C, *future-out* records the components to which C have a future edge pointing, *past-in* and *past-out* record the past edges.

For detailed information, when a transaction T of C initiates its sub-transaction T' of C', C' would remove C from its *future-in* upon receiving message if all its transactions have been already initiated and reply a **ACK_I** message. After C

receives this reply, C might remove C' from *future-out* unless C will use C' in the future. When T' ends, C' sends a **FINISH** message to C. On receiving this message, C adds C' to its *past-out* set and replies a **ACK_F** message. After C' receives this reply, it adds C to its *past-in* set. Thus a component is *quiescent*, *tranquil* or *free* can be specified as an *Alloy* predicate:

```
pred Quiescent (t: Tick,c: Component) {
  Dependent[c] in Passive[t]
  c in Passive[t]
}

pred Freeness (t: Tick,c: Component) {
  !(some FutureIn[t,c] and some PastIn[t,c])
}
pred Tranquil (t: Tick,c : Component) {
  no c': Adjacent[c] |
    c' in FutureIn[t,c] and c' in PastIn[t,c]
}
```

Fig. 17: Specifications of different conditions

C. Verification of DSU protocols

The controllers of DSU protocols only allow the update to happen when the target component has reached the safe state at the update time. In order to check the correctness of each DSU protocols, we assume that any dynamic update will not violate *version consistency* [19], which indicates the global consistency (φ) for DSU protocols. It says that each pair of transactions that share the same component cannot have more than one versions during one process. To simplify verification process, we just check the transactions of the target component since other component's versions always remain unchanged. *Version consistency* and the correctness of DSU protocols can be specified:

```
pred VersionConsistency {
  all disj tx,tx': Update.component.transaction |
    tx.hostVersion=tx'.hostVersion
}

assert DSUCorrectness {
  ControlledSystem and Controllers implies
      VersionConsistency
}
```

Fig. 18: Assertion of correctness of DSU protocols

During verification, a big scope is dispensable yet a small one might be insufficient to uncover potential errors. Unlike the 2PC protocol, there exists some relations between the number of each signature in DSU protocols (explained in IV-B). We set the scope of verification is 3 Components and 4 Transactions for tradeoff. The result of three protocols is shown below.

TABLE III: Result of counterexample

	# Clauses	Time(s)	Counterexample
Quiescence	160751	89.895	No
Tranquillity	341069	791.828	Yes
VC	341001	6383.928	No

Table III shows that controllers of *quiescence* and VC protocols can guarantee global consistency of DSU, and controller of *tranquillity* protocol, however, cannot. The counterexample of *tranquillity* protocol is shown in Fig. 19.

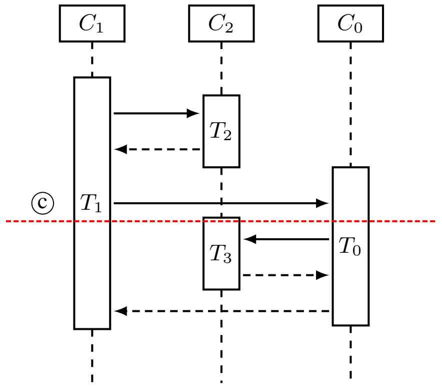


Fig. 19: Counterexample of *tranquillity* protocol

In Fig. 19, we see that T_1 is C_1's transaction, T_0 is C_0's, and T_3 and T_2 are C_2's. We intend to update component C_2 at time ⓒ. At time ⓒ, component C_2 is *tranquil* and thus it is updated. However this update violates *version consistency* since T_2 starts before update with old version and T_3 starts after update with new version. This counterexample illustrates that the specification of controller is correct and that *tranquillity* protocol cannot guarantee global consistency. Some similar and unexpected faults might occur if we misuse *tranquillity* protocol.

The results in Table III and Fig. 19 agree with existed proofs of corresponding protocol [17], [18], [19], which reflects the correctness of controllers. However, from Table III we see that the time of verifying VC protocol is too long even if scope is small. Hence we also use local invariants based modular verification to accelerate it.

D. Local Invariants in VC protocol

In VC protocol, a transaction T cannot be initiated before its *parent-transaction* T' is initiated, *i.e.*, T does not send/receive any messages before T' does, which means their actions are dependent. In order to integrate modular verification with VC protocol, we should split the system into several blocks where each pair of actions is independent. Intuitively, each local part contains exactly two adjacent components since *parent-transaction* and *sub-transaction* does not exist in only two components.

The sufficient condition of VC protocol is whether the target component is *free*, *i.e.*, there is not a pair of *future/past* edges entering it. A future edge $C \rightarrow C'$ implies that C might request a service of C', so this future edge should not be removed until the last initiation of the transaction of C'. Similarly, a past edge $C \rightarrow C'$ implies that C has requested a service of C' before, so this past edge should be added at the end of the first transaction of C'. If we consider the initiation and accomplishment separately, *i.e.*, for each pair of components, and choose l to be the last initiation of all transactions and the first accomplishment of them, P_l will be the action of removing *future* edges and adding *past* edges. For instance in Fig. 20, the past edge will be added as soon as the first transaction is accomplished.

```
pred ManagePastEdge {
  all disj c, c' : Component | {
    some c'.transaction & subTransaction[c.
        transaction] implies {
      let t1 = {
        tick : Tick | noMsgRecBefore[c, c', tick,
            FINISH] and someMsgRecNow[c', c, tick,
            FINISH]
      }, t2 = {
        tick : Tick | noMsgRecBefore[c, c', tick,
            ACK_F] and someMsgRecNow[c, c', tick,
            ACK_F]
      },t3 = Root.endTime.prev | {
        all t : range[t1, t3] | c' in PastOut[t, c]
        all t : range[t2, t3] | c in PastIn[t, c']
      }
    }
  }
}
```

Fig. 20: Predicate of managing past edge in VC protocol

Instead of modeling details of VC protocol, the above rules of managing future and past edges are included in the model. We also assume that it will not violate *Version Consistency* (see Fig. 21). The scope is same as before. The comparison of this modular verification (we call it *VCvalidity*) with former verification (we call it *VCmessage*) is shown in Table IV.

```
assert CorrectnessOfVCinLocalInvariants  {
  (ControlledSystem and ManagePastEdge and
      ManageFutureEdge) implies VersionConsistency
}
```

Fig. 21: Assertion of VC protocol in local invariants

TABLE IV: Comparison of verification time

	#Clauses	Time(s)	Counterexample
VCmessage	341001	6383.928	No
VCvalidity	248103	1206.670	No

Table IV shows that number of clauses in *VCvalidity* is much fewer than *VCmessage* and verifying time of *VCvalidity* is far less than that of *VCmessage*. We extend the comparison to different scopes. In Fig. 22, the size is larger with the increasing of the scope-Id of x-axis. We only consider those scopes whose time is within 24 hours and the largest scope in the comparison is 3 Components and 5 Transactions. From Fig. 22 we see that *VCvalidity* is always more efficient than *VCmessage* and with the increasing of model, the effect is more significant.

These two invariants in our modular verification is same as the concept of *Future-validity* and *Past-validity* in *valid configuration* [19]. The difference is that their proof of the

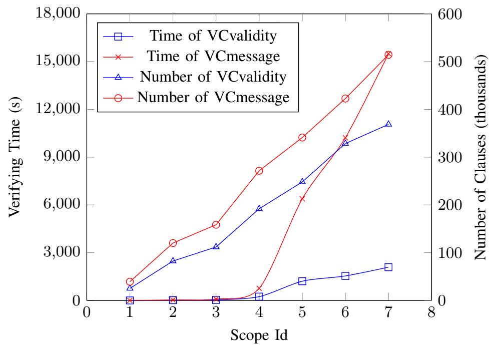


Fig. 22: Comparison of VCvalidity and VCmessage

correctness of VC protocol relies on the assumption of *valid configuration*, whereas our approach can provide formal verification with/without this assumption. Moreover, we show how to obtain them and leverage them for efficient verification.

V. Related work

Although our work mainly focuses on manual controller, controller synthesis has attracted a lot of research. Synthesis from formal declarative specifications has been studied with the aim of providing an operational model to support requirements and analysis [3] on event-based operational models [2] or self-adaptive systems [1], [4]. However, existing techniques for automatic synthesis have limitations [2] that they are designed to work in the context of idealized environment models [4], [22] and they also require complete descriptions of the environment. To address that, the synthesis technique in [23] supports a behavior model in which the controlled actions can fail. In [24], Muscholl leveraged Mazurkiewicz trace theory and Zielonka's theorem for the synthesis of concurrent programs and decentralized runtime monitoring.

Control theory [25] is capturing an increasing interest from the software engineering community that looks at self-adaptation as a means to meet QoS requirements despite unpredictable changes in the execution environment [26], [27], [28]. Filieri and Maggio discussed how control theory results can be integrated in the design of self-adaptive software and provided details on the steps of the control design process [29]. They proposed a closed-loop control strategy that provides formal guarantees for an adaptive software system's dynamic behavior [27] and extended it to support multiple goals [30]. They also introduced a paradigm called brownout, which helps to build more robust cloud applications using control theory [31].

Assume-guarantee reasoning is well known in *compositional reasoning* by verifying each component in isolation [32]. Flanagan *et al.* used assume-guarantee reasoning to ensure the reliability [33] and correctness properties [34] of shared-memory programs. Besides assume-guarantee reasoning, parameterized model checking [35], [36], [37], [38] and partial order reduction [39] are also well known for combating state explosion problems. Although the problem of verification of parameterized systems is undecidable [40], there are two possible remedies to this situation: either we should look for restricted scopes for which the problem becomes decidable [35], [41], [42], or devise methods which are sound but necessarily incomplete, and hope that the system of interest will yield to one of these methods [36], [43], [44]. The idea of our approach corresponds with the former one and the concept of independent actions in our approach is inspired by partial order reduction. In order to reduce the state space of the transition system, partial order reduction attempts to identify path fragments of the full transition system and does not preserve certain properties, while our approach intends to explore unchanged properties of part of the transition system and uses them as assumptions for efficient verification.

Symmetry reduction [45], [46] has also been leveraged to deal with state explosion. Emerson and Sistla [46] exploited symmetry in model checking for concurrent systems composed by many identical or isomorphic components. The basic idea is to reduce model checking over the original structure to model checking over a smaller quotient structure, where symmetric states are identified. Clarke et al. [45] tried to formalize symmetry reduction and identified a class of temporal logic formulas that are preserved under this reduction.

DSU has attracted a lot of related work, in addition to the mentioned three DSU protocols quiescence [17], tranquillity [18], VC [19]. While some work [47] focuses on the architectural changes of the systems, some considers the behavior [48], [49]. Zhang [50] proposed a model-based development process of adaptive programs with finite state machines and LTL specifications. Hayden et al. [51] presented a method of automatically verifying the correctness of dynamic updates of C programs.

VI. Conclusion

In this paper, we give an exercise of verifying distributed controllers with local invariants and use the 2PC protocol and DSU protocols to demonstrate our approach. The result of our controllers in our case study corresponds with existing theoretical analysis. Thus our approach offers a way to formally verify the correctness of corresponding DSU protocols [17], [18], [19], whose original proofs are semi-formal. Besides, after leveraging local invariants, the size of model is reduced and the time of verification is accelerated, with the increasing of the scope, the effect is more significant.

Several issues require further investigations in our future work. First of all, we attempt to generalize a definition of local invariants and give a detailed analysis of the rate of acceleration of verification. Second, in order to figure out a general procedure to facilitate users to select local invariants, we need to experiment our approach on more cases.

Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472177, 91318301, 61321491) of China. We would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[2] N. D’Ippolito, “Synthesis of event-based controllers for software engineering,” Ph.D. dissertation, Imperial College London, 2013.

[3] E. Letier, J. Kramer, J. Magee, and S. Uchitel, “Deriving event-based transition systems from goal-oriented requirements models,” *Automated Software Engineering*, vol. 15, no. 2, pp. 175–206, 2008.

[4] J. Kramer and J. Magee, “Self-managed systems: an architectural challenge,” in *Future of Software Engineering, 2007. FOSE’07.* IEEE, 2007, pp. 259–268.

[5] C. G. Cassandras *et al.*, *Introduction to discrete event systems*. Springer Science & Business Media, 2008.

[6] A. D. Kshemkalyani, M. Raynal, and M. Singhal, “An introduction to snapshot algorithms in distributed computing,” *Distributed systems engineering*, vol. 2, no. 4, p. 224, 1995.

[7] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[8] A. Vakili and N. A. Day, “Temporal logic model checking in alloy,” in *Abstract State Machines, Alloy, B, VDM, and Z*. Springer, 2012, pp. 150–163.

[9] M. Taghdiri and D. Jackson, *A lightweight formal analysis of a multicast key management scheme*. Springer, 2003.

[10] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh, “Formal verification of oauth 2.0 using alloy framework,” in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*. IEEE, 2011, pp. 655–659.

[11] B. Fraikin, M. Frappier, and R. St-Denis, “Supervisory control theory with alloy,” *Science of Computer Programming*, vol. 94, pp. 217–237, 2014.

[12] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, “Evaluating the “small scope hypothesis”,” *Unpublished*, 2003.

[13] O. Grumberg and D. E. Long, “Model checking and modular verification,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 843–871, 1994.

[14] C. B. Jones, “Specification and design of (parallel) programs.” 1983.

[15] A. Pnueli, *In transition from global to modular temporal reasoning about programs*. Springer, 1985.

[16] J. Gray and L. Lamport, “Consensus on transaction commit,” *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.

[17] J. Kramer and J. Magee, “The evolving philosophers problem: Dynamic change management,” *Software Engineering, IEEE Transactions on*, vol. 16, no. 11, pp. 1293–1306, 1990.

[18] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt, “Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 856–868, 2007.

[19] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna, and J. Lu, “Version-consistent dynamic reconfiguration of component-based distributed systems,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 245–255.

[20] O. Maler, A. Pnueli, and J. Sifakis, “On the synthesis of discrete controllers for timed systems,” in *STACS 95*. Springer, 1995, pp. 229–242.

[21] A. Van Lamsweerde and E. Letier, “Handling obstacles in goal-oriented requirements engineering,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 10, pp. 978–1005, 2000.

[22] D. Sykes, W. Heaven, J. Magee, and J. Kramer, “Plan-directed architectural change for autonomous systems,” in *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2007, pp. 15–21.

[23] N. D’Ippolito, V. Braberman, N. Piterman, and S. Uchitel, “Synthesis of live behaviour models for fallible domains,” in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 211–220.

[24] A. Muscholl, “Automated synthesis of distributed controllers,” in *Automata, Languages, and Programming*. Springer, 2015, pp. 11–27.

[25] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback control theory*. Courier Corporation, 2013.

[26] T. Patikirikorala, A. Colman, J. Han, and L. Wang, “A systematic survey on the design of self-adaptive software systems using control engineering approaches,” in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2012, pp. 33–42.

[27] A. Filieri, H. Hoffmann, and M. Maggio, “Automated design of self-adaptive software with control-theoretical formal guarantees,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 299–310.

[28] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, “Control theory for model-based performance-driven software adaptation,” in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, 2015, pp. 11–20.

[29] A. Filieri, M. Maggio, K. Angelopoulos, N. D’Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein *et al.*, “Software engineering meets control theory,” in *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 2015, pp. 71–82.

[30] A. Filieri, H. Hoffmann, and M. Maggio, “Automated multi-objective control for self-adaptive software design,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 2015, pp. 13–24.

[31] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, “Brownout: Building more robust cloud applications,” in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 700–711.

[32] S. Berezin, S. Campos, and E. M. Clarke, *Compositional reasoning in model checking*. Springer, 1998.

[33] C. Flanagan, S. N. Freund, and S. Qadeer, “Thread-modular verification for shared-memory programs,” in *Programming Languages and Systems*. Springer, 2002, pp. 262–277.

[34] C. Flanagan and S. Qadeer, “Assume-guarantee model checking,” Technical report, Microsoft Research, Tech. Rep., 2003.

[35] A. Pnueli, S. Ruah, and L. Zuck, “Automatic deductive verification with invisible invariants,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2001, pp. 82–97.

[36] Y. Kesten and A. Pnueli, “Control and data abstraction: The cornerstones of practical formal verification,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 328–342, 2000.

[37] E. A. Emerson and V. Kahlon, “Parameterized model checking of ring-based message passing systems,” in *Computer Science Logic*. Springer, 2004, pp. 325–339.

[38] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, “Parameterized verification with automatically computed inductive assertions?” in *Computer Aided Verification*. Springer, 2001, pp. 221–234.

[39] P. Godefroid and D. Pirottin, “Refining dependencies improves partial-order verification methods,” in *Computer Aided Verification*. Springer, 1993, pp. 438–449.

[40] K. R. Apt and D. C. Kozen, “Limits for automatic verification of finite-state concurrent systems,” *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, 1986.

[41] S. M. German and A. P. Sistla, “Reasoning about systems with many processes,” *Journal of the ACM (JACM)*, vol. 39, no. 3, pp. 675–735, 1992.

[42] E. A. Emerson and K. S. Namjoshi, “Reasoning about rings,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 85–94.

[43] R. P. Kurshan and K. McMillan, “A structural induction theorem for processes,” in *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. ACM, 1989, pp. 239–247.

[44] D. Lesens, N. Halbwachs, and P. Raymond, “Automatic verification of parameterized linear networks of processes,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1997, pp. 346–357.

[45] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, “Exploiting symmetry in temporal logic model checking,” *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 77–104, 1996.

[46] E. A. Emerson and A. P. Sistla, “Symmetry and model checking,” *Formal methods in system design*, vol. 9, no. 1-2, pp. 105–131, 1996.

[47] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Architecture-based runtime software evolution,” in *Proceedings of the 20th International Conference on Software Engineering*, ser. ICSE ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 177–186.

[48] J. Kramer and J. Magee, “Analysing dynamic change in software architectures: a case study,” in *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, May 1998, pp. 91–100.

[49] S. An, X. Ma, C. Cao, P. Yu, and C. Xu, “An event-based formal framework for dynamic software update,” in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 173–182.

[50] J. Zhang and B. H. Cheng, “Model-based development of dynamically adaptive software,” in *Proceedings of the 28th International Conference on Software Engineering*. ACM, 2006, pp. 371–380.

[51] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster, “Specifying and verifying the correctness of dynamic software updates,” in *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 278–293.