Reusable Requirements in Automated Verification of Distributed Systems

Richard Skowyra rskowyra@bu.edu
Andrei Lapets lapets@bu.edu
Azer Bestavros best@bu.edu
Assaf Kfoury kfoury@bu.edu

Computer Science Department
Boston University

ABSTRACT

The growing popularity of infrastructure-as-a-service cloud computing, software-defined networking, and related technologies have enabled the rapid creation of complex, large-scale distributed systems. Many of these systems are used by applications with stricter requirements than those covered by SLAs, such as those used by the financial, healthcare, and industrial sectors. Mathematical methods exist which can be used to formally verify many of these safety, liveness, and security properties, but are rarely used by system designers.

In this paper, we identify *brittle requirements* as one of the problems which impede the use of formal methods in distributed system design, and propose a solution based on the decomposition of a formal model into a user-defined component and one or more domain abstractions. This decomposition enables reusable requirements, which can be shared across models without requiring redefinition or remapping of variable bindings. We provide a network-based example of domain abstraction, and define reusable requirements over this abstraction in several well-known logics. We concretely implement model decomposition with VML, a lightweight modeling language based on labeled transition systems. VML models can be used to rapidly prototype new distributed systems by utilizing domain abstractions with reusable requirements. To demonstrate, we create two examples using imperative and symbolic VML models.

1. INTRODUCTION

The economic success of cloud computing and the burgeoning industry of Big Data has led to a profusion of large-scale mission-critical distributed systems. Many of these systems are being used for applications in finance, healthcare, and other industries which require iron-glad guarantees with respect to safety, liveness, security, and privacy. These are currently addressed largely through Service-Level Agreements (SLAs), which impose a contractual obligation to meet specific requirements established by the customer. SLAs, however, are not proofs that a requirement is always met. Rather, they are legally binding penalties for the provider failing to meet that requirement.

For applications where SLAs are insufficient, formal mathematical methods exist which can verify (with quantifiable certainty) that a guarantee holds over a system or service. These techniques check a set of requirements (expressed as formulas in one of more logics) over all reachable states of a model, attempting to discover counter-examples where a requirement is not met.

Unfortunately, formal verification is rarely used in practice outside of very high cost-of-failure scenarios like integrated circuit design. Due to their generally high algorithmic complexity, they are also much more tractable when applied to simplified models of a system, such as those used in the design phase.

The rarity of formal verification in the design of distributed systems is attributed to a number of factors [34, 24]. While some of these are addressed in recent tools and techniques [42, 16], we identify *brittle requirements* as an ongoing problem which significantly limits the adoption of formal verification as a system design tool. Generally, formal models are created in order to check whether certain requirements in that system's specification (e.g., availability, consistency, and safety properties) are indeed met.

These requirements are often inseparable from the model itself, leading to three forms of brittleness that we have observed: specificity, parameter-varying syntax, and local verification. These are discussed in Section 2 in more detail, but all lead to requirements that cannot be shared between models despite being broadly applicable (e.g., the impossibility of packet reordering), or are so tightly bound to the model that minor changes to its implementation necessitate substantial changes to the requirement syntax. Requirement brittleness makes incremental or iterative model development difficult at best, and often limits the viability of using formal methods during the design and prototyping phase, when the system is evolving rapidly. Unfortunately, this phase is precisely where such techniques would be most useful and tractable.

To address these issues and to make formal verification more accessible to designers of distributed systems, we present VML, an extensible modeling language which implements an abstraction-based mechanism for separating models and their requirements. Currently, VML provides a Network Domain Abstraction that represents a network of concurrent hosts communicating asynchronously via message-passing. This simple abstraction is sufficient to enable the expression of many critical properties of distributed systems, from the presence of forwarding loops or network blackholes to packet loss, re-ordering and duplication. In Section 5 we provide an example list of formally defined, useful properties.[1] Any of these properties can be checked against any VML model which utilizes their associated domain abstraction, without any transformation or re-writing to be done.

In the future we intend to extend VML with other Domain Abstractions, including software-defined networking, replication, and consistency models. Using these abstractions, VML enables libraries of re-usable properties to be built and shared among the research community, independently of the models being investigated by individual researchers.

VML is a first attempt at a language-based implementation of a more general model design framework for the separation of a model and its properties. In this framework, a model is decomposed into two parts: a series of opaque Domain Abstractions which can be modified by the user only via an exported interface, and the user-written model code. Done correctly, a large class of interesting properties can be defined over only the protected Domain Abstractions, which enables requirement invariance and re-use both across different versions of the same model and across entirely different models. This decomposition resembles the separation of an operating system into user-space and kernel-space code, and should embody several of the same characteristics: strict but movable separation of the two domains, controlled isolation of the state of the protected domain from changes made by the user domain, and separate development of user-domain and protected-domain implementations (i.e., of models, Domain Abstractions, and their associated properties).

The remainder of the paper is as follows. In Section 2 we elaborate on the problems discussed above, which impede the adoption of formal modeling techniques in distributed system design. In Section 3 we propose a model-decomposition design framework for formal modeling which addresses the problems identified in Section 2. In Section 4 we present VML, a language-based implementation of the model-decomposition technique. In Section 5, we provide a list of formal properties and corresponding high-level English descriptions which are useful for formal reasoning about distributed systems. In Section 6, we model example systems in VML, and discuss the interaction of the user-generated model with the network abstraction. We present related work in Section 7, discuss future work in Section 8, and conclude in Section 9.

2. MOTIVATING EXAMPLES

In this section, we describe the process of formal verification and three kinds of brittleness in requirement specification that impede its adoption as a design tool for distributed systems. Each problem is illustrated using implementations of existing, well-known formal models written for a particular verification environment.

Formal verification is an automated search for counter-examples to logically defined requirements over a finite state space of possible model states. The model is generally written by the user in a tool-specific language like Promela [12], Alloy [15], SAL [35], etc. These languages often serve as a way to specify instances of particular formalisms, such as Büchi Automata, Boolean Formulas, or Abstract State Machines.

Once a model is written, requirements to be checked are defined using a requirement specification language. In some tools (e.g., Alloy) this is the same language used to write the model, while in others (e.g., SPIN) it is separate language or logic.

The verification tool automatically composes the model and its requirements to form a state space of possible model configurations. The space is exhaustively searched for reachable states in which one or more requirements fail to hold. These are then presented to the user.

This approach is lightweight compared to theorem-proving and proof assistants, in that it requires little initial work and verification is accomplished quickly, but it sacrifices completeness. While incomplete verification is problematic when used to prove that a property holds in all cases, it is quite useful at the design phase to quickly discard designs where a requirement demonstrably does not hold.[2] More attention can then be devoted to stronger designs, over which a future step of theorem-proving may be employed if a higher level of confidence is necessary.

Unfortunately, the benefits of formal verification are often outweighed by cumbersome, inflexible modeling and requirement specification languages. We identified three particular forms of brittleness which cause difficulty in defining reusable requirements that can be shared across models. Each is described in detail below, with examples drawn from models written with the SPIN and PRISM verification tools. These were chosen due to an abundance of available models, and relevance to the modeling of distributed systems, but the issues we discuss are not local to these tools.

Due to the low level of abstraction supported by the majority of formal modeling languages, requirements must often be specified in such a way as to be tightly bound to model syntax and structure. This is problematic if a model is being developed iteratively, or if a system is being studied in a variety of configurations, etc. We have observed three particularly prominent kinds of this brittleness in requirements specification:

Specificity. Requirements may be bound to specific, statically defined model states, array indices, and other hard-coded values (e.g., $\texttt{local\_state} == 3 \;\wedge\; \texttt{ar}[5] == \texttt{true}$). Minor model updates can easily invalidate properties by removing or changing these.

Specificity is particularly prevalent in models based on communicating state machines [27, 26, 28, 29, 31, 30, 33, 25, 23]. These models are often written in an extremely low-level syntax, where models are essentially just primitive data elements and a transition function governing valid updates. Some languages do provide a facility to map unbound predicates and atoms to model-specific data. This allows the logical syntax of a requirement to be re-used over

[1]Note that this list is not intended to be exhaustive. The abstraction restricts only the data elements usable by logical formulas, not the logic being used or the formula syntax.

[2]The small-scope hypothesis [1] posits that this incompleteness is less problematic than it would first appear. Significant empirical evidence indicates that most real-world bugs (as opposed to maliciously designed flaws) which are detected in large-scale, time-intensive verification, are also present at much smaller, more easily verified scales.

related models, but variables and predicates must still be re-mapped by hand [32, 28, 8].

Parameter-Varying Syntax. Requirements may be defined in such a way that modifying a model parameter (e.g., the number of processes in a system) requires the rewriting of multiple requirements. Many of these rewrites are linear in the size of the parameter (e.g., $p_1 \wedge p_2 \wedge ... \wedge p_n$) [32, 27, 26, 28, 29, 31, 20, 17, 36]. Certain properties based on combinations of model elements (e.g., $(p_1 \wedge p_2) \vee (p_1 \wedge p_3) \vee (p_2 \wedge p_3)$ require substantially more rewriting, however. The requirements defined in both [30] and [18] scale quadratically, for example. We observed that many of these examples exist due to the necessity of specifying universally or existentially quantified requirements in unquantified specification logics.

Many instances of parameter-varying syntax can be solved by writing a pre-processor which consumes a parameterized version of a model and updates the requirement specification appropriately. The pre-processor is model-specific, however, and it is unreasonable to expect a system designer to write one for every model under development.

Localized Verification. Requirements may be defined as assertions or predicates which must (not) hold true at certain regions of model code. Adding new code or modifying those regions may introduce false positives or negatives when that requirement is verified, or may require model-refactoring to preserve correctness. Models written in imperative languages like Promela [8, 43, 44] most frequently encounter this problem.

3. MODEL DECOMPOSITION

In this section, we consider how a formal model and requirements to be checked over it can be separated from one another in order to enable the reuse of requirements across different models, as well as the automatic binding of these requirements to a specific model. Our approach relies on model decomposition, a design technique inspired by operating systems and the network protocol stack.

As demonstrated in Section 2, formal models are very tightly coupled to the requirements that are being checked. This makes sense for application-specific properties which are meaningful only with respect to that model, but becomes problematic when a common high-level property of a domain (e.g., whether packets are dropped in a network) must be checked.

We propose to decouple these two aspects of formal verification via model decomposition, a design technique in which a formal model is decomposed into a user-defined component and one or more Domain Abstractions (DAs). These are defined formally below. Informally, DAs syntactically encapsulate common semantic structures within a given application domain. Each DA exposes operations to the user in the form of new a syntax augmenting the core language, VML, described in section 4. As an ongoing example, the Network Abstraction that we define in Section 3.2 allows formal modeling of those network protocols that can limit their network interaction to a set of seven operations: send, receive, forward, copy, link, unlink, and message typing.

Syntactic encapsulation of common semantics enables large classes of requirements to be defined completely independently of the model over which they are being checked, including before or after model development. We show in Section 5 that the classes of requirement which benefit from this separation (specifically, those logical formulas whose

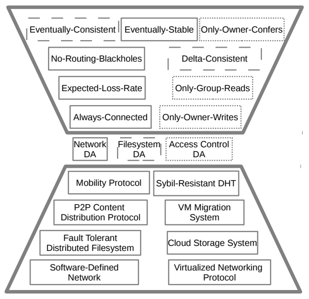

Figure 1: Domain Abstraction as a narrow waist

variables map only to data elements in the abstraction) are quite wide-ranging and include many common requirements of distributed systems.

Done correctly, encapsulation of common structures via Domain Abstractions can create a *narrow-waist* in the model design space. Below this waist, a wide variety of models can be written using syntax exposed by Domain Abstractions. Above it, a wide variety of requirements, in any specification language desired, can be defined against the formal structure of Domain Abstractions. Any of these requirements can be checked against any model using the relevant DA.

Furthermore, the design of Domain Abstractions, and the method by which they are composed with one another, should acknowledge that the boundary between abstraction and detail is flexible, domain-specific, and prone to change. To this end, it should be possible to extend the set of abstractions with refinements, varying levels of detail, and new structures developed by the application domain's community. We were inspired in this design by the modular definition of kernel- and user-space processes in the operating systems community: the core infrastructure should remain minimal and stable, while permitting new modules and functionalities to be dynamically loaded or removed from the kernel space.

The Network Abstraction, for example, does not include the ability to define properties related to hop-count. This is not because hop-count is difficult to model or a special case, but simply because many systems will not need it and should not be forced to model it. Other DAs could certainly be defined which do include it, and many other interesting properties such as notions of consistency or concurrency could be encapsulated in Domain Abstractions. System designers should be able to use and compose any number of these abstractions in order to best support the system under study.

3.1 Domain Abstractions

Having given an informal description of model decomposition and Domain Abstractions, we now formally define what constitutes a DA and the requirements imposed in order to automate their composition.

DEFINITION 1. *A Domain Abstraction is a Labeled Transition System $\mathcal{L} = (S, A, \rightarrow, I)$, where:*

- *S is a set of states;*
- *A is a set of actions;*
- *$\rightarrow \subseteq S \times A \times S$ is a ternary transition relation;*
- *I is a set of initial states.*

Domain Abstractions are synchronously composed with an initial LTS defined by the user. In order to facilitate this composition, one additional requirement is imposed on the LTS: I and $\rightarrow$ must be augmented with a *stutter* transition which, for all $s \in S$, enables the LTS to remain in the same state pre- and post-transition. This is necessary to prevent deadlocks in states where the user-defined model does not transition using a synchronizing syntax (see below), that is, when it has only internal state transitions or makes a transition exposed by another DA.

Domain Abstractions which may be manipulated interactively (i.e., do not consist purely of static data) have a further requirement: at least one $a \in A$ must have an associated *synchronizing syntax*, a (possibly parameterized) language element which is exposed to the core language and may be included in the user-defined model. The synchronizing syntax is used to synchronously compose the user-defined model with the DA: when the former transitions using the exposed syntax, the latter transitions from the current state $s \in S$ to $s' \in S$ such that $\exists a \in A, (s, a, s') \in \rightarrow$. If there exist multiple possible post-states, the choice is made nondeterministically. Any parameterization of the synchronizing syntax is treated as a constraint which must hold over s'. If no s' exists for the the current state, action, and constraint, the system will deadlock.

3.2 The Network Domain Abstraction

In this section, we formally define the network domain abstraction, or *network abstraction*. We first present it as a finite state space with transitions defined as symbolic constraints over pre- and post- states, then provide a straightforward algorithm to construct a labeled transition system from the constrained state space.

Definition 2. *A* network abstraction *is defined to be an 8-tuple $\mathcal{N} = (H, M, T, S, \rho, \sigma, \tau, \delta)$; each component of the tuple is defined below.*

1. H, M, and T are static, finite, non-empty atomic sets. H denotes the set of hosts, M denotes the set of messages, and T denotes the set of message types; there always exists a bottom type $\perp \in T$ (undefined).

2. S denotes the state space of the abstraction. As explained below, $S \subseteq \rho \times \sigma \times \tau$.

3. $\rho : H \times H$ is a binary relation between hosts, such that $(h_i, h_j) \in \rho$ or $\rho(h_i, h_j)$ denotes a one-way connection from h_i to h_j.

4. $\sigma : M \times H$ is a binary relation between a message and a host such that $(m_i, h_i) \in \sigma$ or $\sigma(m_i, h_i)$ denotes the presence of message m_i in the set of received (but not yet processed) messages at host h_i.

5. $\tau : M \times T$ is a typing function that maps messages to either a user-defined message type or the bottom type $\perp$.

6. $\delta : S \times S$ is a symbolic transition function defined over the state space, detailed below.

The cardinality of the sets H, M, and T are user-defined parameters that, once set, remain constant for the entire verification process. This fixes a state space $S_0 \subseteq \rho \times \sigma \times \tau$ where each relation is a set of tuples whose maximum size is determined by the cardinality of H, M, and T. We further restrict the state space to all states in which τ is a total function, that is, in which all messages are typed:

$$S \quad = \quad \{s \mid s \in S_0 \text{ and } \forall m \in M, \exists t \in T \text{ s.t. } \tau(m, t)\}$$

We define a symbolic transition function δ as the union of pre- and post-state constraints over the network abstraction. Each constraint corresponds to an action in the network, which is represented as a change in the valuations of one or more relations. For all $s, s' \in S$ (where s, s' are tuples of the form $\rho \times \sigma \times \tau$ and $\rho' \times \sigma' \times \tau'$, respectively), the transition $(s, s') \in \delta$ is included if for any of the following constraints, s satisfies the pre-state constraint and s' satisfies the post-state constraint. Note that only the state of the binary relations may differ in the pre- and post-states. Membership in the sets H, M, and T does not change.

$$\text{ORIG}\;\frac{\begin{array}{c}\exists h_i, h_j \in H, \rho(h_i, h_j) \\ \exists m \in M, \forall h_k \in H, \neg\sigma(m, h_k) \\ \exists t \in T\end{array}}{\rho' = \rho \quad \sigma' = \sigma \cup \{(m, h_j)\} \quad \tau' = \tau \cup \{m \mapsto t\}} \tag{1}$$

$$\text{RCV}\;\frac{\exists h \in H, m \in M, t \in T \text{ s.t. } \sigma(m, h) \wedge \tau(m, t)}{\sigma' = \sigma - \{(m, h)\} \quad \rho' = \rho \quad \tau' = \tau} \tag{2}$$

$$\text{FWD}\;\frac{\exists h_i, h_j \in H, m \in M \text{ s.t. } \sigma(m, h_i) \wedge \rho(h_i, h_j)}{\sigma' = \sigma - \{(m, h_i)\} \cup \{(m, h_j)\} \quad \rho' = \rho \quad \tau' = \tau} \tag{3}$$

$$\text{COPY}\;\frac{\exists h_i, h_j \in H, m \in M \text{ s.t. } \sigma(m, h_i) \wedge \rho(h_i, h_j)}{\sigma' = \sigma \cup \{(m, h_j)\} \quad \rho' = \rho \quad \tau' = \tau} \tag{4}$$

$$\text{LINK}\;\frac{\exists h_i, h_j \in H \text{ s.t. } \neg\rho(h_i, h_j)}{\sigma' = \sigma \quad \rho' = \rho \cup \{(h_i, h_j)\} \quad \tau' = \tau} \tag{5}$$

$$\text{UNLINK}\;\frac{\exists h_i, h_j \in H \text{ s.t. } \rho(h_i, h_j)}{\sigma' = \sigma \quad \rho' = \rho - \{(h_i, h_j)\} \quad \tau' = \tau} \tag{6}$$

$$\text{RETYPE}\;\frac{\exists t \in T \quad \exists m \in M \text{ s.t. } \forall h \in H, \neg\sigma(m, h)}{\sigma' = \sigma \quad \rho' = \rho \quad \tau' = \tau \cup \{m \mapsto t\}} \tag{7}$$

$$\text{STUTTER}\;\frac{}{\sigma' = \sigma \quad \rho' = \rho \quad \tau' = \tau} \tag{8}$$

Each of the above constraints corresponds to one of the actions below that can be taken by the network.

- **ORIGINATE** types and sends a new message to a host as long as a connection to that host exists.

- **RECEIVE** removes (processes) a message from the set of messages in flight to a host.

- **FORWARD** corresponds to receiving and re-transmitting a message. Note that this differs from a **RECEIVE** followed by an **ORIGINATE**, as in the former case no constraint is put on the number of message copies, while the latter is valid only if no copies of the message are extant in the network.

- **COPY** duplicates a message from one host to another, as long as a connection to that host exists.

- **LINK** and **UNLINK** add and remove a connection between two hosts, respectively.

- **RETYPE** changes the type of a message, as long as that message is not currently in flight over the network.

- **STUTTER** allows the network to remain the same in its pre- and post-state.

We formally verified the consistency of the above abstraction in Alloy for multiple cardinalities of H, M, and T, and confirmed in all cases that all constraints are satisfiable over state space S.

THEOREM 1. *It is possible to construct a nondeterministic labeled transition system $\mathcal{L}_{\mathcal{N}} = (S', A', \rightarrow, I')$ from any network abstraction $\mathcal{N} = (H, M, T, S, \rho, \sigma, \tau, \delta)$.*

PROOF. Construct the restricted state space S as above, given H, M, and T. Set $S' = S$ and $I' = S'$. Enumerate each constraint c that defines δ, and create an associated action in $c \in A'$. In order to construct $\rightarrow$, it is necessary to identify for each $s \in S'$ every possible post-state $s' \in S'$ and create the appropriately labeled transition. An algorithm to do so is straightforward to define:

```
for $s \in S'$ do
    for each constraint $c$ in the definition of $\delta$ do
        if $s$ satisfies the pre-condition of $c$ then
            for $s' \in S'$ do
                if $s'$ satisfies the post-condition of $c$ then
                    $\rightarrow \ := \ \rightarrow \cup \{(s, c, s')\}$
                end if
            end for
        end if
    end for
end for
```

$\square$

The translation of the symbolic transition system to a labeled transition system may create a state explosion during verification due to the number of transitions in $\rightarrow$. As will be shown in Section 4.2, this problem is mitigated by the synchronous composition of the network abstraction with a user-defined host model defined using VML.

Table 1 presents the VML syntax for specifying individual actions that can be taken by the network abstraction. Note that $\mathcal{A}$ denotes the set of all possible actions for a network abstraction. This syntax is incorporate into the host model VML syntax described in Section 4.1 and defined in Table 2.

natural n, i, j	$\in$	$\mathbb{N}$
host h	$\in$	H
message m	$\in$	M
message type t	$\in$	$\mathcal{T}$
action a	$::=$	$\texttt{originate}(m, t, h_i, h_j)$
	$\mid$	$\texttt{receive}(h, m, t)$
	$\mid$	$\texttt{forward}(m, h_i, h_j)$
	$\mid$	$\texttt{copy}(m, h_i, h_j)$
	$\mid$	$\texttt{link}(h_i, h_j)$
	$\mid$	$\texttt{unlink}(h_i, h_j)$
	$\mid$	$\texttt{retype}(m, t)$
	$\mid$	$\texttt{stutter}()$
set of all actions $\mathcal{A}$	$::=$	$\{a \mid a \text{ is an action}\}$

Table 1: Network abstraction VML syntax.

4. VML

VML is a lightweight modeling language designed to permit rapid, iterative development of specifications. VML makes it possible to integrate formal verification into the design phase of a distributed system

4.1 Host Model Composed with Network Abstraction

A *host model* specifies the behavior of hosts within a network. The VML syntax for specifying the host model is presented in Table 2. The behavior of a host is modelled as a finite state space and associated transition function. The state space represents all possible states of variables (global and local to each host), the possible positions of control flow in the host model specification for each host, and all the possible states of the imported network abstraction; valid transitions are determined by the semantics of the statements (in terms of control flow, modifications to local and global variable state, and modifications to the network abstraction's representation of the network state) specified by the user within each host definition in the host model.

DEFINITION 3. *Given an existing network abstraction denoted $\mathcal{N} = (H, M, T, S, \rho, \sigma, \tau, \delta)$ with a set of hosts $H = \{h_1, \ldots, h_n\}$ and a set of actions $\mathcal{A}$, a* host model *is a tuple $\mathcal{H} = (P \times S,\ \pi)$; the component S is imported from $\mathcal{N}$; the components P and π of the tuple are defined below.*

1. $P = G \times L_{h_1} \times \ldots \times L_{h_n}$ is the set of all possible distinct states of the collection of hosts:

    (a) G is the set of possible states of the collection of global variables;

    (b) $L_{h_1}, \ldots, L_{h_n}$, where each set L_i is the set of possible states of the local variables and local control flow in a particular host (each L_h may differ in structure depending on the VML host definition corresponding to that host $h \in H$).

2. π is a transition function in $(P \times S) \times \mathcal{A} \times (P \times S)$; valid transitions in this function are determined by the statements in each host definition and the conditions under which they can be executed:

    (a) any update to the position of the control flow and/or to the local or global variable state from $p \in P$ to $p' \in P$ in a given host definition's body does not alter the network abstraction state $s \in S$ and corresponds to the **STUTTER** action in the network abstraction:

    $$((p, s), \texttt{stutter}(), (p', s)) \in \pi;$$

    (b) any update to the network abstraction state from $s \in S$ to $s' \in S$ via an explicit statement invoking action a in a given host definition's body corresponds to the appropriate action in the network abstraction, and it is also necessary to update the control flow position and (e.g., if a host received a message and stored it locally in a named variable) the local variable state from $p \in P$ to $p' \in P$:

    $$((p, s), a, (p', s')) \in \pi.$$

THEOREM 2. *It is possible to construct a nondeterministic labeled transition system $\mathcal{L}_{\mathcal{H}} = (S', A', \rightarrow, I')$ from any host model $\mathcal{H} = (P \times S, \pi)$.*

PROOF. Let $A' = \mathcal{A}$, let $S' = P \times S$ and $I' = S'$, and let $\rightarrow = \pi$. $\square$

natural k, n	$\in$	$\mathbb{N}$
variable x	$\in$	$\mathcal{V}$
constant c	$\in$	$\mathcal{K}$
action a	$\in$	$\mathcal{A}$
net. abs. $\mathcal{N}$		
rich ident. y	$::=$	x_1 `.` $\ldots$ `.` x_n
	$\mid$	$\mathcal{N}$`.H` $\mid$ $\mathcal{N}$`.M` $\mid$ $\mathcal{N}$`.T`
	$\mid$	$\mathcal{N}$`.rho` $\mid$ $\mathcal{N}$`.tau` $\mid$ $\mathcal{N}$`.sigma`
	$\mid$	$\mathcal{N}$`.link`
term e	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$
	$\mid$	`true` $\mid$ `false` $\mid$ y $\mid$ c $\mid$ $*$
	$\mid$	e_1 `&&` e_2 $\mid$ e_1 `││` e_2 $\mid$ `!`e
	$\mid$	e_1 `==` e_2 $\mid$ e_1 `<` e_2 $\mid$ e_1 `<=` e_2
	$\mid$	$-e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \cdot e_2$
	$\mid$	`│` e `│`
	$\mid$	`(` e_1 `,` $\ldots$ `,` e_n `)`
	$\mid$	e_1 `(` e_1 `,` $\ldots$ `,` e_n `)`
	$\mid$	`{` e_1 `,` $\ldots$ `,` e_n `}`
branch b	$::=$	e `:` $s_1 \ldots s_n$
	$\mid$	`else:` $s_1 \ldots s_n$
statement s	$::=$	$\mathcal{N}$`.`a e
	$\mid$	e_1 `=` e_2
	$\mid$	e `:` $s_1 \ldots s_n$
	$\mid$	`if :` $b_1 \ldots b_n$
	$\mid$	`loop :` $s_1 \ldots s_n$
host h	$::=$	`host` x `:` $s_1 \ldots s_n$
import $\mathcal{I}$	$::=$	`import` $\mathcal{N}$
model $\mathcal{M}$	$::=$	$\mathcal{I}$ `init:` $s_1 \ldots s_n$ $h_1 \ldots h_n$

Table 2: Host model VML syntax.

4.2 Restriction of Network Domain Abstraction by the Host Model

Given an instance of a network abstraction $\mathcal{N}$ and an instance of a host model $\mathcal{H}$, it is possible to restrict the network abstraction so that its transition relation only contains transitions permitted by the host model. It is then possible to interpret any requirement that applies to $\mathcal{N}$ as a requirement that applies to the composition of $\mathcal{N}$ and $\mathcal{H}$.

DEFINITION 4. *A* restriction *of any network abstraction $\mathcal{N} = (H, M, T, S, \rho, \sigma, \tau, \delta)$ by a host model $\mathcal{H} = (P \times S, \pi)$ is denoted $\mathcal{N}|\mathcal{H}$, and is defined as $(H, M, T, S|\mathcal{H}, \rho, \sigma, \tau, \delta|\mathcal{H})$ where:*

$$s \in S|\mathcal{H} \quad \text{iff} \quad \exists p, p', s' \text{ s.t. } ((p, s), a, (p', s')) \in \pi$$
$$(s, s') \in \delta|\mathcal{H} \quad \text{iff} \quad \exists p, p' \text{ s.t. } ((p, s), a, (p', s')) \in \pi$$

THEOREM 3. *Given any existing network abstraction $\mathcal{N} = (H, M, T, S, \rho, \sigma, \tau, \delta)$ and host model $\mathcal{H} = (P \times S,\ \pi)$, any formula φ defining a requirement over a network abstraction $\mathcal{N}$ can be checked over the restricted network abstraction $\mathcal{N}|\mathcal{H} = (H, M, T, S|\mathcal{H}, \rho, \sigma, \tau, \delta|\mathcal{H})$. In other words, if it is possible to check whether $\mathcal{N} \vdash \varphi$, then it is possible to check whether $\mathcal{N}|\mathcal{H} \vdash \varphi$.*

PROOF. This follows from the fact that $S|\mathcal{H} \subseteq S$ and $\delta|\mathcal{H} \subseteq \delta$. $\square$

5. EXPRESSING FORMAL PROPERTIES OF DISTRIBUTED SYSTEMS

Given a logical system Φ, a property or requirement for a network abstraction $\mathcal{N}$ is a formula $\varphi \in \Phi$ such that it is possible to determine using an automated process whether it is true that $\mathcal{N} \vdash \varphi$. In this section, we provide examples of properties which may be defined over Domain Abstractions and thereby re-used between models. We concentrate on the Network Domain Abstraction, but the principles presented here apply equally to any formalism conforming to the definition of Domain Abstraction presented in Section 3.

The ability to express a requirement is dependent on two factors: the population of available predicates and variables (what can be checked), and the formal language used to create formulas over that population (what can be asked). Domain Abstractions restrict only the former, allowing new and future languages and logics to express requirements over existing models. In order to formally verify these requirements, of course, a verification tool supporting the given logic and with a modeling language that an LTS can be translated to must be available. We are currently developing translators from VML to Promela, Alloy, and PRISM (see Section 8).

Domain Abstractions restrict the set of available variables and predicates by exposing a finite set of atomic variables or relations, which collectively constitute the internal state of the abstraction. Any property (regardless of formal language used) which can be expressed purely in terms of this internal state can be separated from individual models. Recall that the network abstraction exposes four such relations: $\rho : H \times H$, $\sigma : M \times H$, $\tau : M \times T$, and $\rightarrow \subset S \times A \times S$. The first three denote links between network hosts, messages in flight to hosts, and message typings, respectively. The fourth is the transition relation over the state space S and set of actions A.

5.1 Sample Requirements

Below, we provide a small sample of requirements which may be defined over the network abstraction, using First-Order Logic (FOL) and Linear Temporal Logic (LTL). Both have substantial verification support, and have been used in the past to reason about distributed systems. These samples are not intended to be in any way exhaustive, but are merely intended to serve as examples of the expressive power that simple abstractions can still provide.

Before continuing, recall that all requirements are being checked over a finite state space. This allows universal and existential quantification to be reduced to the simple intersection or union, respectively, of all atomic elements in the set being quantified. [14]. First-Order Logic is therefore reducible to propositional calculus, and may be checked with a SAT solver. In the case that temporal or other logical quantifiers are also used, FOL sub-formulas may be replaced by their propositional equivalent (e.g., $\mathcal{G}(\forall x \exists y\ x = y)$ can be rewritten as:

$$\mathcal{G}((x_0 = y_0 \vee x_0 = y_1 \ldots) \wedge (x_1 = y_0 \vee x_1 = y_1 \ldots) \wedge \ldots).$$

5.1.1 First-Order Logic

First-order logic provides universal and existential quantification over sets and relations. This enables the expression of many useful requirements pertaining to the current state of a system, especially with respect to models which study

Requirement	Formula	Logic
Bidirectional-Links	$\forall h \forall h'\ \rho(h,h') \rightarrow \rho(h',h)$	FOL
All-Inflight-Messages-Are-Broadcast	$\forall m \exists h\ \sigma(m,h) \rightarrow \forall h'\ \sigma(m,h')$	FOL
No-Messages-Duplicated	$\forall m \exists h \exists h' \sigma(m,h) \wedge \sigma(m,h') \rightarrow h = h'$	FOL
No-Untyped-Messages-Inflight	$\forall m \forall h \sigma(m,h) \rightarrow \neg\tau(m,\bot)$	FOL
One-Outgoing-Link	$\forall h \exists h' \exists h''\ \rho(h,h') \wedge \rho(h,h'') \rightarrow h' = h''$	FOL
Star-Topology	$\exists h \forall h'\ \rho(h',h) \wedge \rho(h,h') \wedge \forall h''\ h'' \neq h \rightarrow \neg\rho(h',h'')$	FOL
All-Hosts-Reachable	$\forall h \forall h'\ \rho^+(h,h') \vee h = h'$	FO(TC)
No-Structural-Forwarding-Loops	$\forall h\ \neg\rho^+(h,h)$	FO(TC)
Ring-Topology	All-Hosts-Reachable $\wedge$ One-Outgoing-Link	FO(TC)
No-Cycles	$\forall h\ \neg\rho^+(h,h)$	FO(TC)

Figure 2: Network Domain Requirements in FOL and FO(TC).

static snapshots of possible system configurations. Since there is no explicit notion of time, it is difficult to express properties relating to the reachability of states.

Note that FOL can be extended with a transitive closure operation over binary relations, which enables the expression of reachability predicates. We use X^+ to denote the transitive closure of a binary relation X. This augmentation is strictly more expressive than FOL, and results in the logic called Transitive Closure Logic, or FO(TC). Since we are working in a finite domain, however, FO(TC) remains reducible to a Boolean formula [14]. Figure 5.1.1 presents sample requirements which can be defined in FOL an FO(TC).

5.1.2 *Linear Temporal Logic*

Linear Temporal Logic (LTL) is a popular, well-supported logic for checking concurrency, recurrence, stability, and reachability properties of formal models. LTL is defined over a single infinite path through through a state-space, i.e., an infinite sequence of model states. It enables the expression of temporal properties relating the current and future model state.

The syntax of LTL consists of the standard propositional variables, boolean logical operators, and the temporal operators $\mathcal{X}$ $\mathcal{G}$, $\mathcal{F}$, $\mathcal{U}$, $\mathcal{R}$. The first three are unary that operators bind a formula to the next state, all future states, and some future state, respectively. The last two are binary operators which require the first formula to hold true at least until the second holds true (which must eventually occur), and require the second formula to hold true until and including the point at which the first becomes true, respectively. A full introduction to Linear Temporal Logic can be found in the literature [13].

Note that in addition to the requirements below, any of the FOL formulas above can be used with temporal operators (e.g., $\mathcal{GF}$All-Hosts-Reachable). This, again, is due to the reduction of FOL to Propositional Calculus when quantifying over a finite domain. In fact, since the Network Domain Abstraction consists purely of relations, this quantification is necessary to avoid the use of constants which may not be defined in all models (e.g., $\sigma(m_3, h_4)$).

The easy expression of recurrence and stability properties makes LTL particularly useful. Recurrence requirements take the form $\mathcal{GF}\phi$, and state the fact that in all states, there exists a reachable future state in which ϕ will be true. Stability requirements take the form $\mathcal{FG}\phi$, and state that there is some future reachable state after which ϕ will hold in all states. Figure 5.1.2 presents Network DA requirements which can be expressed in LTL.

Requirement	Formula
All-Sent-Messages-Eventually-Received	$\mathcal{G}(\forall m \exists h\ \sigma(m,h) \rightarrow \mathcal{F}(\neg\sigma(m,h)))$
Host-Disconnects-Once-Messages-Processed	$(\exists h \exists h' \rho(h,h') \mathcal{U} \forall m \neg\sigma(m,h)$
All-Hosts-Eventually-Join	$\mathcal{F}(\forall h \exists h' \rho(h,h') \vee \rho(h',h))$
All-Hosts-Eventually-Leave	$\mathcal{F}(\forall h \forall h' \neg\rho(h,h') \wedge \neg\rho(h',h))$
No-Logical-Forwarding-Loops	$\mathcal{G}(\forall m \exists h \sigma(m,h) \rightarrow \mathcal{F} \forall h' \neg\sigma(m,h'))$

Figure 3: Network Domain Requirements in LTL.

5.2 Inexpressible Requirements

In this section, we discuss what properties cannot be expressed in the current Network Domain Abstraction. These fall into two categories: requirements which use data elements not present in the abstraction, and requirements which use constants not safely addressable in the abstraction. The former consists of, e.g., hop count, source and destination address, message reordering, and message dropping. All of these could be added to the abstraction with minimal difficulty, and are currently not present in an effort to keep the DA as minimal as possible. Hop count, source, and destination address can easily be included as additional relations with corresponding constraints in the symbolic transition function. Checking for message dropping can be expressed in terms of σ and destination address. Message reordering could be expressed by, for example, creating a partial ordering over the σ relation. All of these capabilities can easily be added to the Domain Abstraction, and may be as part of our future work.

The second set of inexpressible requirements, those which use unsafe constants, arises from the parameterization of the Network DA. Recall that the domain and range of σ, ρ, and τ are determined by the user-specified cardinality of the sets H, M, and T. Any formula which binds a variable to a specific atomic element (e.g., $\mathcal{F}\sigma(m_4, h_3)$) is only syntactically correct if that element is a member of its respective set for that model. This prevents generic reuse of such properties across models. Note that other domain absractions may safely use constants, as long as the set in question is invariant across models.

6. VML MODELING EXAMPLES

In this section, we present a two small VML models in order to demonstrate the capability of our model decomposition approach. The first, a simple client and server, demonstrates an imperative modeling style and the use of a reusable LTL property. The second is a constraint-satisfaction problem which finds a network topology obeying certain restrictions. It demonstrates symbolic modeling and the use of a reusable FO(TC) property.

We also describe (but do not include) the implementation of a network of learning switches using the OpenFlow software-defined networking architecture. This larger model uses an expanded VML syntax, which builds on the language kernel presented here and provides convenience functions such as set comprehension and finite iteration. For each model, we discuss possible requirements and note those which can be automatically reused across models (i.e., constrain only states in the domain abstraction).

6.1 Simple Client-Server

Figure 6.1 presents a simple client-server model, in which a client sends requests to a server that receives the request and then replies with a response. It is intended to illustrate

```
import network
init:
  network.H = {CLIENT, SERVER}
  network.M = {M0, M1}
  network.T = {REQUEST, RESPONSE}
  network.sigma = {}
  network.rho = {}
  network.tau = {}

  network.link(CLIENT, SERVER)
  network.link(SERVER, CLIENT)

host Client
  loop:
    network.originate(M0, REQUEST, CLIENT, SERVER):

    network.receive(CLIENT, M1, RESPONSE):

host Server
  loop:
      network.receive(SERVER, M0, REQUEST):
        network.originate(M1, RESPONSE, SERVER, CLIENT)
```

Figure 4: Client-Server Model in VML

```
import network
init:
  network.H = 10
  network.M = 1
  network.T = 1
  //No connection between these hosts is possible
  network.rho(1,2) = 0
  network.rho(2,1) = 0
  network.rho(4,5) = 0
  network.rho(5,4) = 0
  network.rho(5,2) = 0
  network.rho(2,5) = 0
  //These hosts can only support limited outgoing connections
  |network.rho(3,*)| = 2
  |network.rho(1,*)| = 1
  //These hosts can only support limited incoming connections
  |network.rho(*,1)| = 3
  |network.rho(*,7)| = 3
```

Figure 5: Topology-Finding in VML

basic VML modeling and how the user-defined model interacts with the Network Domain Abstraction (NDA). The initialization phase constructs an initial NDA state consisting of two mutually linked hosts and no in-flight messages. All elements of the sets H, M, and T are explicitly named, and the relations σ, ρ, and τ begin empty. The `host` keyword denotes concurrent processes, which will be composed asynchronously. The `loop` keyword indicates that the scoped block will be repeated infinitely. Each process communicates using the NDA's synchronizing syntax. The client non-deterministically chooses to send or receive a message on each iteration.

The client-server model can be checked for deadlocks by translating the composed LTS to a Büchi automaton and verifying using SPIN [12] or a related model checker. Using a Büchi automaton also enables automatic checking of the `All-Sent-Messages-Eventually-Received` property (among others), which is expressed in LTL.

6.2 Model-Finding

The client-server model above demonstrates an imperative modeling technique, in which a user-defined model explicitly specifies state transitions. However, it is also possible to model system state symbolically via constraint satisfaction. This approach, used by tools like the Alloy Analyzer [15], allows instances of a model to be found which satisfy (or not) a set of constraints. This is useful when reasoning about assumptions and invariants in a distributed system without needing to model specific protocol states.

Figure 6.2 presents an example in which certain specific hosts in a network cannot connect to one another, and certain hosts can only support a limited number of connections. If the designer wants to find an instance in which both these constraints and the `All-Hosts-Reachable` requirement hold, the model can be symbolically model-checked via translation to a SAT instance and given to a SAT solver. The resulting satisfying valuation can then be translated back to an instance of the original model [14].

In this model, there are no independent processes to compose with the Network Domain Abstraction. Instead, the initialization phase is used to constrain the NDA by underspecifying its state. Since the names of specific state elements are unimportant, H, M, and T are initialized with their cardinalities rather than a list of symbolic constants. The ρ relation is constrained in two ways: by explicitly setting certain pairs of hosts to be disconnected, and by setting the maximum number of times certain hosts can appear in the domain or co-domain of the relation. The syntax used above is shorthand for a cardinality constraint on the set comprehensions $\{(h)|\exists h' \rho(h,h')\}$ and $\{(h)|\exists h' \rho(h',h)\}$, respectively.

6.3 Software-Defined Networking

For brevity and ease of explanation, the two models above are necessarily limited in scope and complexity. An expanded VML syntax, however, has been used to model and formally reason about software-defined networks. In [37], we modeled an OpenFlow-based network of learning switches and formally verified safety, stability, and reliability properties of the network.

OpenFlow [22] networks outsource routing logic to a domain-specific software controller written by the network designer, which runs on a remote machine (ranging from commodity hardware to custom FPGAs) connected to each switch via a secure channel. OpenFlow-enabled switches send unknown or unhandled packets to this controller, which responds by installing flow-rules (next-hop rules which trigger based on packet headers) in one or more of switches across the network. Compliant switches then route data-plane packets based on these flow rules.

We created VML-based formal models of OpenFlow switches, an OpenFlow controller, and mobile end-hosts which non-deterministically join, send messages to one another, and leave the network. These were used to model a network of learning switches with state centralized in the controller.

We defined five requirements over the network, four of which are written in LTL and one which is written in PCTL*, a linear- and branching-time logic with probabilistic quantifiers:

1. *no-forwarding-loops*: Any packet that enters the network will eventually exit the network.
2. *no-blackholes*: Any packet that is sent will eventually be received.
3. *stable-correct-receiver*: If all nodes cease being mobile, eventually all packets that are received will be received by the intended recipient.
4. *stable-no-floods*: If all nodes cease being mobile, eventually no more packets will be flooded.

5. *bounded-loss-rate*: The expected packet loss rate of mobile nodes is below a specified bound.

Unfortunately, these requirements are not re-usable across models. We are currently implementing an OpenFlow switch Domain Abstraction, however, in order to enable re-usability of these and other useful requirements.

7. RELATED WORK

In this section, we discuss related work over three dimensions: integration techniques for formal models; requirement reuse, domain abstraction, and model decomposition; and formal modeling tools for distributed systems.

VML and composable domain abstractions are intended to allow integration of multiple domain-specific formal models into a single formalism, which can be translated to multiple verification tools dependent on the requirements being verified. Prior work on the integration of formal models has focused largely on either integrating disparate formal languages under a common semantics [39, 38, 6, 7, 41], or developing (semi)automated techniques to transform a formal model in one language into another [3, 21, 40].

These techniques are complementary to ours: domain abstraction enables the composition of multiple models in a single formalism (labeled transition systems), and benefits from both allowing more models to be expressed in that formalism, and from allowing that formalism to be translated to different formal languages for verification.

To the best of the authors' knowledge, requirement reuse has no direct analogue in the research community. The interface-based compositional approach underlying requirement reuse, however, is similar to work in the software engineering community.

Bayley and Zhu [2] formalize design patterns as statements in first-order logic, which can then be composed with one another and automatically verified with respect to requirements over the software model. This is similar to our approach of domain abstraction to enable the automatic composition of domain-specific semantic constructs, which can then be formally verified with respect to system requirements.

Gurov and Huisman [11] present an interface-based compositionality framework for reasoning about the safety and security properties of smartphone applications. Their technique is based on inlining of private functions to safely transform models into public, composable interfaces between software components.

As discussed in Section 8, VML translators are in development for multiple formal verification tools. We intend to utilize VML as a common input language for integrating multiple existing off-the-shelf verification systems into an extensible verification suite. This is similar to the approach taken by Veritech [10], a translation framework for model descriptions. Veritech uses an intermediate Core Design Language (CDL) as common basis for translating a model between several different formal languages. This allows models to be verified under multiple formalisms, which permits wider classes of properties to be verified and for smaller, more tractable verifications by carefully selecting which translator is most appropriate for a given model and set of specifications to check. Unlike Veritech, however, VML's translation facilities are designed to allow a wide variety of pre-made domain abstractions to be composed with a small user-defined model, and verified with respect to reusable formal properties.

MODEST [4] and its tool suite MOTOR [5] is an integrated reasoning environment for reactive systems. A model is written in the MODEST stochastic process algebra, which is designed to capture a number of functional and non-functional system properties including safety, liveness, reliability, and performance. The MOTOR tool uses a single-formalism, multi-solution approach: MODEST models are distilled, based on the property to be checked, into one or more simpler formalisms that can be formally verified using any tool whose input language accepts that formalism. While this enables a larger variety of requirements to be checked, no facility is provided for the reuse of requirements over multiple MOTOR models.

CADP [9] is a formal verification suite based on the theory of concurrent distributed processes. Models are written in one of several supported process calculi, which enable formal reasoning about concurrency, dynamic data structures, and equivalence relations. CADP supports multiple back-end verification systems, including model checking and equivalence checking. There is no concept in CADP of model decomposition or reusable requirements, but we are currently considering implementation of a translator from VML to CADP in order to combine our reusable requirements with its substantial verification capabilities.

8. FUTURE WORK

In this section, we discuss future work in two directions: creating translation algorithms from VML to off-the-shelf verification tools, and creating new Domain Abstractions to enable the expression of new reusable requirements for distributed systems.

With respect to the former direction, we are currently building translators to Promela [12] (the input language of the SPIN explicit-state model checker), Alloy [15](a symbolic model-finder), and PRISM [19] (a probabilistic model checker). SPIN excels at verifying requirements expressible in LTL, as well as finding deadlock and livelock states. Alloy is useful for verifying properties expressible in FO(TC) and relational calculus. Finally, PRISM can verify properties written in PCTL*, a superset of linear time logic and computation tree logic that is augmented with probabilistic quantifiers. We chose these three tools due to their strong research communities and useful, expressive requirement specification logics.

In addition to building VML translators, we are also creating new Domain Abstractions. Software-Defined Networking and Cloud Computing enable flexible, tailor made distributed systems. These systems must often have strong gaurentees on performance, reliability, and correctness. To this end, we are developing a domain abstraction of an OpenFlow switch that will allow requirements to be formally verified over software-defined OpenFlow networks. We also intend to create abstractions useful for modeling cloud computing scenarios, including object replication and consistency.

Finally, we intend to create libraries of useful, formally defined requirements over Domain Abstractions. These reusable requirement libraries will allow designers to concentrate on modeling their system, and not on formally expressing its requirements.

9. CONCLUSION

In this work, we defined a model-decomposition framework which enables reusable requirement specifications for formal models of distributed systems. These requirements can be checked against any model written using a particular Domain Abstraction, without the need to change the requirements' syntax or variable bindings. We formally defined Domain Abstractions, showed they can be represented using Labelled Transition Systems, and proved that any abstraction meeting our definition can be composed with other Domain Abstractions, as well as user-defined models.

To demonstrate this technique we created and formally specified a Network Domain Abstraction, which can be used to reason about network topologies and message passing. We also created a list of sample requirements that can be used, unchanged, in any model which utilizes the network domain abstration. Finally, we defined a modeling language, VML, for writing user-defined models that are automatically composable with any Domain Abstraction. We provided examples of imperative and symbolic VML models which use our Network Domain Abstraction, and discussed what re-usable requirements can be checked over each example.

10. REFERENCES

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

[2] I. Bayley. Formalising design patterns in predicate logic. In *SEFM*, pages 25–36, 2007.

[3] D. Bert and F. Cave. Construction of finite labelled transition systems from b abstract systems. In *Integrated Formal Methods*, pages 235–254. Springer, 2000.

[4] H. Bohnenkamp, P. d'Argenio, H. Hermanns, and J. Katoen. Modest: A compositional modeling formalism for hard and softly timed systems. *Software Engineering, IEEE Transactions on*, 32(10):812–830, 2006.

[5] H. Bohnenkamp, H. Hermanns, J. Katoen, and R. Klaren. The modest modeling tool and its implementation. *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 116–133, 2003.

[6] R. Boute. Integrating formal methods by unifying abstractions. In *Integrated Formal Methods*, pages 441–460. Springer, 2004.

[7] Y. Chen and Z. Liu. Integrating temporal logics. In *Integrated Formal Methods*, pages 402–420. Springer, 2004.

[8] D. Farago. Model checking of randomized leader election algorithms. Master's thesis, Karlsruhe Institute of Technology.

[9] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: a toolbox for the construction and analysis of distributed processes. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387, 2011.

[10] O. Grumberg and S. Katz. Veritech: a framework for translating among model description notations. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(2):119–132, 2007.

[11] D. Gurov and M. Huisman. Interface abstraction for compositional verificatio. In *SEFM*, pages 414–424, 2005.

[12] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2005.

[13] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*, volume 2. Cambridge University Press Cambridge" UK, 2004.

[14] D. Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000.

[15] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, Apr. 2002.

[16] C. Jaspan, M. Keeling, L. Maccherone, G. Zenarosa, and M. Shaw. Software mythbusters explore formal methods. *Software, IEEE*, 26(6):60–63, 2009.

[17] M. Kwiatkowska and G. Norman. Verifying randomized Byzantine agreement. In D. Peled and M. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *LNCS*, pages 194–209. Springer, 2002.

[18] M. Kwiatkowska, G. Norman, and D. Parker. Analysis of a gossip protocol in prism. *ACM SIGMETRICS Performance Evaluation Review*, 36(3):17–22, 2008.

[19] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *23rd International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[20] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. Springer, 2001.

[21] S. Marrone, C. Papa, and V. Vittorini. Multiformalism and transformation inheritance for dependability analysis of critical systems. In *Integrated Formal Methods*, pages 215–228. Springer, 2010.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[23] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[24] D. Parnas. Really rethinking'formal methods'. *Computer*, 43(1):28–34, 2010.

[25] T. R. P.R. D'Argenio, J-P. Katoen and J. Tretmans. he bounded retransmission protocol must be on time! In *3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97, Enschede, The Netherlands*.

[26] PRISM. Itai and rodeh asynchronous leader election protocol.

[27] PRISM. Itai and rodeh synchronous leader election protocol.

[28] PRISM. Randomised dining philosophers: Lehmann and rabin.

[29] PRISM. Randomised dining philosophers: Lynch, saias and segala.

[30] PRISM. Randomised mutual exclusion: Pnueli and zuck.

[31] PRISM. Randomised mutual exclusion: Rabin.

[32] PRISM. Randomised self-stabilising algorithms.

[33] PRISM. Randomised two process wait-free test-and-set: Tromp and vitanyi.

[34] W. Schulte. Why doesn't anyone use formal methods? In *Integrated Formal Methods*, pages 297–298. Springer, 2000.

[35] N. Shankar. Combining theorem proving and model checking through symbolic analysis. *CONCUR 2000âĂŤConcurrency Theory*, pages 1–16, 2000.

[36] G. Singh and S. K. Shukla. Verifying compiler based refinement of bluespectm. In *SPIN*, pages 250–269, 2008.

[37] R. Skowyra, A. Lapets, A. Bestavros, and A. Kfoury. Verifiably-Safe Software-Defined Networks for CPS. In *Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems (HiCoNS 2013)*, Philedelphia, PA, USA, April 2013.

[38] G. Smith. An integration of real-time object-z and csp for specifying concurrent real-time systems. In *Integrated Formal Methods*, pages 267–285. Springer, 2002.

[39] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of object-z and csp. *Formal Methods in System Design*, pages 249–284, 2001.

[40] D. Thivolle and H. Garavel. Verification of gals systems by combining synchronous languages and process calculi. In *SPIN*. Citeseer, 2009.

[41] T. Willemse. Embeddings of hybrid automata in process algebra. In *Integrated Formal Methods*, pages 343–362. Springer, 2004.

[42] J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[43] P. Zave. Understanding sip through model-checking. *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*, pages 256–279, 2008.

[44] P. Zave and J. Rexford. Compositional network mobility.