An Inheritance-Based Technique for Building Simulation Proofs Incrementally

Idit Keidar, Roger Khazan, Nancy Lynch,
MIT Lab for Computer Science
545 Technology Sq., Room 367
Cambridge, MA 02139, USA
{idish, roger, lynch}@theory.lcs.mit.edu
+1 617 253 1922

Alex Shvartsman
University of Connecticut
Computer Science and Engineering Dept.
Storrs, CT 06269-3155, USA
aas@cse.uconn.edu
+1 860 486 2672

ABSTRACT

This paper presents a technique for incrementally constructing safety specifications, abstract algorithm descriptions, and *simulation proofs* showing that algorithms meet their specifications.

The technique for building specifications (and algorithms) allows a child specification (or algorithm) to inherit from its parent by two forms of incremental modification: (a) *interface extension*, where new forms of interaction are added to the parent's interface, and (b) *specialization* (subtyping), where new data, restrictions, and effects are added to the parent's behavior description. The combination of interface extension and specialization constitutes a powerful and expressive incremental modification mechanism for describing changes that do not override the behavior of the parent, although it may introduce new behavior.

Consider the case when incremental modification is applied to both a parent specification S and a parent algorithm A. A proof that the child algorithm A' implements the child specification S' can be built incrementally upon a simulation proof that algorithm A implements specification S. The new work required involves reasoning about the modifications, but does not require repetition of the reasoning in the original simulation proof.

The paper presents the technique mathematically, in terms of automata. The technique has already been used to model and validate a full-fledged group communication system (see [26]); the methodology and results of that experiment are summarized in this paper.

Keywords

System modeling/verification, simulation, refinement, specialization by inheritance, interface extension.

1 INTRODUCTION

Formal modeling and validation of software systems is a major challenge, because of their size and complexity. Among the factors that could increase widespread usage of formal methods is improved cost-effectiveness and scalability (cf. [20, 22]). Current software engineering practice addresses problems of building complex systems by the use of incremental development techniques based on an object-oriented approach. We believe that successful efforts in system modeling and validation will also require incremental techniques, which will enable reuse of models and proofs.

In this paper we provide a framework for reuse of proofs analogous and complementary to the reuse provided by object-oriented software engineering methodologies. Specifically, we present a technique for incrementally constructing safety specifications, abstract algorithm descriptions, and *simulation proofs* that algorithms meet their specifications. Simulation proofs are one of the most important techniques for proving properties of complex systems; such proofs exhibit a *simulation relation* (*refinement mapping*, *abstraction function*) between a formal description of a system and its specification [13, 24, 29].

The technique presented in this paper has evolved with our experience in the context of a large-scale modeling and validation project: we have successfully used this technique for modeling and validating a complex group communication system [26] that is implemented in C++, and that interacts with two other services developed by different teams. The group communication system acts as *middleware* in providing tools for building distributed applications. In order to be useful for a variety of applications, the group communication system provides services with diverse semantics that bear many similarities, yet differ in subtle ways. We have modeled the diverse services of the system and validated the algorithms implementing each of these services. Reuse of models and proofs was essential in order to make this task feasible. For example, it has allowed us to avoid repeating the five-page long correctness proof of the algorithm that provides the most basic semantics when proving the correctness of algorithms that provide the more sophisticated semantics. The correctness proof of the most sophisticated algorithm, by comparison, was only two and a half pages long. (We describe our experience in this project as well as the methodology that evolved from it in Section 6.)

Our approach to the reuse of specifications and algorithms through inheritance uses incremental modification to derive a new component (specification or algorithm), called *child*, from an existing component called *parent*. Specifically, we present two constructions for modifying existing components:

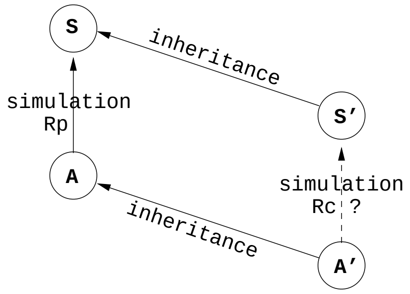
1. We allow the child to *specialize* the parent by reusing its state in a read-only fashion, by adding new state components (read/write), and by constraining the set of behaviors of the parent. This corresponds to the *subtyping* view of inheritance [8]. We will show that any observable behavior of the child is *subsumed* (cf. [1]) by the possible behaviors of the parent, making our specialization analogous to the *substitution inheritance* [8]. In particular, the child can be used anywhere the parent can be used. (Specialization is the subject of Section 3.)

2. A child can also be derived from a parent by means of *interface (signature) extension*. In this case the state of the parent is unchanged, but the child may include new observable actions not found in the parent and new parameters to actions that exist at the parent. When such new actions and parameters are hidden, then any behavior of the child is exactly as some behavior of the parent. (Interface extension is presented in Section 5.)

When interface extension is combined with specialization, this corresponds to the *subclassing for extension* form of inheritance [8] which provides a powerful mechanism for incrementally constructing specifications and algorithms. Consider the following example. The parent defines an unordered messaging service using the send and recv primitives. To produce a totally ordered messaging service we specialize the parent in such a way that recv is only possible when the current message is totally ordered. Next we introduce the safe primitive, which informs the sender that its message was delivered. First we extend the service interface to include safe primitives and then we specialize to enable safe actions just in case the message was actually delivered.

The specialization and extension constructs can be applied at both the specification level and the algorithm level in a way that preserves the relationship between the specification and the algorithm. The main technical challenge addressed in this paper (in Section 4) is the provision of a formal framework for the reuse of simulation proofs especially for the specialization construct. Consider the example in Figure 1: Let S be a specification, and A an abstract algorithm description. Assume that we have proven that A *implements* S using a simulation relation $\mathtt{R_p}$. Assume further that we specialize the specification S, yielding a new *child* specification S'. At the same time, we specialize the algorithm A to construct an algorithm A' which supports the additional semantics required by S'.

Figure 1 Algorithm A simulates specification S with $\mathtt{R_p}$. Can $\mathtt{R_p}$ be reused for building a simulation $\mathtt{R_c}$ from a child A′ of A to a child S′ of S?



When proving that A' implements S', we would like to rely on the fact that we have already proven that A implements S, and to avoid the need to repeat the same reasoning. We would like to reason only about the new features introduced by S' and A'. The proof extension theorem in Section 4 provides the means for incrementally building simulation proofs in this manner.

Simulation proofs [13] lend themselves naturally to be supported by interactive theorem provers. Such proofs typically break down into many simple cases based on different actions. These can be checked by hand or with the help of interactive theorem provers. Our incremental simulation proofs break down in a similar fashion.

We present our incremental modification constructs in the context of the I/O automata model [30, 32] (the basics of the model are reviewed in Section 2). I/O automata have been widely used in formulating formal service definitions and abstract implementations, and for reasoning about them, e.g., [6, 9, 11, 12, 14, 15, 21, 24, 28, 31]). An important feature of the I/O automaton formalism is its strong support of composition. For example, Hickey et al. [24] used the compositional approach for modeling and verification of certain modules in Ensemble [19], a large-scale, modularly structured, group communication system. Introducing inheritance into the I/O automaton model is vital in order to push the limits of such projects from verification of individual modules to verification of entire systems, as we have experienced in our work on such a project [26]. Furthermore, a programming and modeling language based on I/O automata formalism, IOA [17, 18] has been defined.

We intend to exploit the IOA framework, to develop IOA-based tools to support the techniques presented in this paper both for validation and for code generation.

Stata and Guttag [36] have recognized the need for reuse in a manner similar to that suggested in this paper, which facilitates reasoning about correctness of a subclass given the correctness of the superclass is known. They suggest a framework for defining programming guidelines and supplement this framework with informal rules that may be used to facilitate such reasoning. However, they only address informal reasoning and do not provide the mathematical foundation for formal proofs. Furthermore, [36] is restricted to the context of sequential programming and does not encompass reactive components as we do in this paper.

Many other works, e.g., [1, 6, 10, 23, 25, 33], have formally dealt with inheritance and its semantics. Our distinguishing contribution is the provision of a mathematical framework for incremental construction of simulation proofs by applying the formal notion of inheritance at two levels: specification and algorithm.

2 TECHNICAL BACKGROUND

This section presents background on the I/O automaton model, based on [30], Ch. 8. In this model, a system component is described as a state-machine, called an *I/O automaton*. The transitions of the automaton are associated with named actions, classified as *input*, *output* and *internal*. Input and output actions model the component's interaction with other components, while internal actions are externally unobservable.

Formally, an I/O automaton `A` consists of: an interface (or signature), `sig(A)`, consisting of input, output and internal actions; a set of states, `states(A)`; a set of start states, `start(A)`; and a state-transition relation (a subset of `states(A)` $\times$`sig(A)` $\times$`states(A)`), `trans(A)`.

An action π is said to be *enabled* in a state `s` if the automaton has a transition of the form (`s`, π, `s'`); input actions are enabled in every state. An *execution* of an automaton is an alternating sequence of states and actions that begins with a start state, and successive triples are allowable transitions. A *trace* is a subsequence of an execution consisting solely of the automaton's external actions. The I/O automaton model defines a *composition operation* which specifies how automata interact via their input and output actions.

I/O automata are conveniently presented using the *precondition-effect* style. In this style, typed state variables with initial values specify the set of states and the start states. Transitions are grouped by action name, and are specified using a `pre:` block with preconditions on the states in which the action is enabled and an `eff:` block which specifies how the pre-state is modified. The effect is executed *atomically* to yield the post-state.

Simulation Relations

When reasoning about an automaton, we are only interested in its externally-observable behavior as reflected in its traces. A common way to specify the set of traces an automaton is allowed to generate is using (abstract) I/O automata that generate the legal sets of traces. An implementation automaton satisfies a specification if all of its traces are also traces of the specification automaton. Simulation relations are a commonly used technique for proving trace inclusion:

Definition 2.1 *Let* `A` *and* `S` *be two automata with the same external interface. Then a relation* `R` $\subseteq$ `states(A)` $\times$ `states(S)` *is a* simulation from `A` to `S` *if it satisfies the following two conditions:*

1. *If* `t` *is any initial state of* `A`*, then there is an initial state* `s` *of* `S` *such that* `s` $\in$ `R(t)`*.*
2. *If* `t` *and* `s` $\in$ `R(t)` *are reachable states of* `A` *and* `S` *respectively, and if* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* `A`*, then there exists an execution fragment of* `S` *from* `s` *to* $\mathtt{s}'$ *having the same trace, and with* $\mathtt{s}' \in \mathtt{R}(\mathtt{t}')$*.*

The following theorem emphasizes the significance of simulation relations. (It is proven in [30], Ch. 8.)

Theorem 2.1 *If* `A` *and* `S` *are two automata with the same external interface and if* `R` *is a simulation from* `A` *to* `S` *then* `traces(A)` $\subseteq$ `traces(S)`*.*

The simulation relation technique is *complete*: any finite trace inclusion can be shown by using simulation relations in conjunction with history and prophecy variables [2, 35].

3 SPECIALIZATION

Our *specialization* construct captures the notion of subtyping in I/O automata in the sense of trace inclusion; it allows creating a child automaton which specializes the parent automaton. The child can read the parent's state, add new (read/write) state components, and restrict the parent's transitions. The *specialize* construct defined below operates on a parent automaton, and accepts three additional parameters: a *state extension* – the new state components, an *initial state extension* – the initial values of the new state components, and a *transition restriction* which specifies the child's addition of new preconditions and effects (modifying new state components only) to parent transitions. We define the specialization construct formally below.

Definition 3.1 *Let* `A` *be an automaton; let* `N` *be a set of states, called a state extension; let* $\mathtt{N}_0$ *be a non-empty subset of* `N`*, called an initial state extension; let*

$\mathtt{TR} \subseteq (\mathtt{states(A)} \times \mathtt{N}) \times \mathtt{sig(A)} \times \mathtt{N}$ *be a relation, called a transition restriction. For each action* π, $\mathtt{TR}$ *specifies the additional restrictions that a child places on the states of* $\mathtt{A}$ *and* $\mathtt{N}$ *in which* π *is enabled and specifies how the new state components are modified as a result of a child taking a step involving* π.

Then $\mathtt{specialize(A)(N, N_0, TR)}$ *defines an automaton* $\mathtt{A}'$ *as follows:*

- $\mathtt{sig(A')} = \mathtt{sig(A)}$;
- $\mathtt{states(A')} = \mathtt{states(A)} \times \mathtt{N}$;
- $\mathtt{start(A')} = \mathtt{start(A)} \times \mathtt{N_0}$;
- $\mathtt{trans(A')} = \{ (\langle \mathtt{t_p}, \mathtt{t_n} \rangle, \pi, \langle \mathtt{t'_p}, \mathtt{t'_n} \rangle) \mid (\mathtt{t_p}, \pi, \mathtt{t'_p}) \in \mathtt{trans(A)} \wedge (\langle \mathtt{t_p}, \mathtt{t_n} \rangle, \pi, \mathtt{t'_n}) \in \mathtt{TR} \}$

Notation 3.2 *If* $\mathtt{A}' = \mathtt{specialize(A)(N, N_0, TR)}$ *we use the following notation: Given* $\mathtt{t} \in \mathtt{states(A')}$, *we write* $\mathtt{t}|_\mathtt{p}$ *to denote its parent component and* $\mathtt{t}|_\mathtt{n}$ *to denote its new component. If* α *is an execution sequence of* $\mathtt{A}'$, *then* $\alpha|_\mathtt{p}$ ($\alpha|_\mathtt{n}$) *denotes a sequence obtained by replacing each state* $\mathtt{t}$ *in* α *with* $\mathtt{t}|_\mathtt{p}$ ($\mathtt{t}|_\mathtt{n}$). *We also extend this notation to sets of states and to sets of execution sequences.*

We now exemplify the use of the specialization construct. Figure 2 presents a simple algorithm automaton, WRITE_THROUGH_CACHE, implementing a sequentially-consistent register $\mathtt{x}$ shared among a set of processes $\mathtt{P}$. Each process $\mathtt{p} \in \mathtt{P}$ has access to a local $\mathtt{cache_p}$. Register $\mathtt{x}$ is initialized to some default value $\mathtt{v_0}$. A $\mathtt{write_p(v)}$ request propagates $\mathtt{v}$ to both $\mathtt{x}$ and $\mathtt{cache_p}$. A response $\mathtt{read_p(v)}$ to a read request returns the value $\mathtt{v}$ of $\mathtt{p}$'s local $\mathtt{cache_p}$ without ensuring that it is current. Thus, a process $\mathtt{p}$ responds to a read request with a value of $\mathtt{x}$ which is at least as current as the last value previously seen by $\mathtt{p}$ but not necessarily the most up-to-date one.

Figure 3 presents an atomic write-through cache automaton, ATOMIC_WRITE_THROUGH_CACHE, as a specialization of WRITE_THROUGH_CACHE. The specialized automaton maintains an additional boolean variable $\mathtt{synched_p}$ for each process $\mathtt{p}$ in order to restrict the behavior of the parent so that a response to a read request returns the latest value of $\mathtt{x}$. The traces of this automaton are indistinguishable from those of a system with a single shared register and no cache.

In general, the transition restriction denoted by this type of precondition-effect code is the union of the following two sets:

- All triples of the form $(\mathtt{t}, \pi, \mathtt{t}|_\mathtt{n})$ for which π is not mentioned in the code for $\mathtt{A}'$, i.e., $\mathtt{A}'$ does not restrict transitions involving π. The $\mathtt{read\_req_p}$ action of Figure 2 is an example of such a π. Note that the new state component, $\mathtt{t}|_\mathtt{n}$, is not changed.
- All triples $(\mathtt{t}, \pi, \mathtt{t'_n})$ in which state $\mathtt{t}$ satisfies new preconditions on π placed by $\mathtt{A}'$ and in which state $\mathtt{t'_n}$ is the result of applying π's new effects to $\mathtt{t}$.

Figure 2 Write-through cache automaton.

AUTOMATON WRITE_THROUGH_CACHE

Signature:
```
Input:    write_p(v)
          read_req_p()
Output:   read_p(v)
Internal: synch_p()
```
State:
```
x ← v_0
(∀ p ∈ P) cache_p ← x
(∀ p ∈ P) req_p ← 0
```
Transitions:
```
INPUT write_p(v)
eff: x ← v
     cache_p ← v

INTERNAL synch_p()
eff: cache_p ← x

INPUT read_req_p()
eff: req_p ← req_p + 1

OUTPUT read_p(v)
pre: req_p > 0
     v = cache_p
eff: req_p ← req_p - 1
```

Figure 3 Atomic write-through cache automaton.

AUTOMATON ATOMIC_WRITE_THROUGH_CACHE
MODIFIES WRITE_THROUGH_CACHE

State Extension:
```
(∀ p ∈ P) Bool synched_p, initially true
```
Transition Restriction:
```
INPUT write_p(v)
eff: (∀ q ∈ P) synched_q ← false

INTERNAL synch_p()
eff: synched_p ← true

OUTPUT read_p(v)
pre: synched_p = true
```

Theorem 3.1 below says that every trace of the specialized automaton is a trace of the parent automaton. In Section 4, we demonstrate how proving correctness of automata presented using the specialization operator can be done as incremental steps on top of the correctness proofs of their parents.

Theorem 3.1 *If* $\mathtt{A}'$ *is a child of an automaton* $\mathtt{A}$, *then:*

1. $\mathtt{execs(A')}|_\mathtt{p} \subseteq \mathtt{execs(A)}$.
2. $\mathtt{traces(A')} \subseteq \mathtt{traces(A)}$.

Proof 3.1:

1. Straightforward induction on the length of the execution sequence. Basis: If $\mathtt{t} \in \mathtt{start(A')}$, then

$\texttt{t}|_\texttt{p} \in \texttt{start(A)}$ by the definition of $\texttt{start(A}')$. Inductive Step: If $(\texttt{t}, \pi, \texttt{t}')$ is a step of $\texttt{A}'$, then $(\texttt{t}|_\texttt{p}, \pi, \texttt{t}'|_\texttt{p})$ is a step of A, by the definition of $\texttt{trans(A}')$.

2. Follows from Part 1 and the fact that $\texttt{sig(A}') = \texttt{sig(A)}$. Alternatively, notice that trace inclusion is implied by Theorem 2.1 and the fact that the function that maps a state $\texttt{t} \in \texttt{states(A}')$ to $\texttt{t}|_\texttt{p}$ is a simulation mapping from $\texttt{A}'$ to A. ■

4 INCREMENTAL PROOFS

The formalism we have introduced allows not only for code reuse, but also, as we show in this section, for proof reuse by means of incremental proof construction. We start with an example, then we prove a general theorem.

An Example of Proof Reuse

We now revisit the shared register example of Section 3. We present a parent specification of a sequentially-consistent shared register, and describe a simulation that proves that it is implemented by the WRITE_THROUGH_CACHE automaton presented in the previous section. We then derive a child specification of an atomic shared register by specializing the parent specification. Finally, we illustrate how a proof that automaton ATOMIC_WRITE_THROUGH_CACHE implements the child specification can be constructed incrementally from the parent-level simulation proof.

Figure 4 presents a standard specification of a sequentially-consistent shared register x. The interface of SEQ_CONSISTENT_REGISTER is the same as that of WRITE_THROUGH_CACHE. The specification maintains a sequence hist-x of the values stored in x during an execution. A $\texttt{write}_\texttt{p}(\texttt{v})$ request appends v to the end of hist-x. A response $\texttt{read}_\texttt{p}(\texttt{v})$ to a read request is allowed to return *any* value v that was stored in x since p last accessed x; this nondeterminism is an innate part of sequential consistency. The specification keeps track of these last accesses with an index $\texttt{last}_\texttt{p}$ in the hist-x.

We argue that automaton WRITE_THROUGH_CACHE of Figure 2 satisfies this specification by exhibiting a simulation relation R. R relates a state t of WRITE_THROUGH_CACHE to a state s of SEQ_CONSISTENT_REGISTER as follows:

```
(t, s) ∈ R  ⟺
   last(s.hist-x) = t.x
   ∧ (∀ p ∈ P) (∃ hi_p ∈ Integer) such that
               1 ≤ s.last_p ≤ hi_p ≤ |s.hist-x|
               ∧  s.hist-x(hi_p) = t.cache_p
   ∧ (∀ p ∈ P) s.req_p = t.req_p
```

Let (t, s) ∈ R. A step of WRITE_THROUGH_CACHE initiating from state t and involving $\texttt{read}_\texttt{p}(\texttt{v})$ simulates a step of SEQ_CONSISTENT_REGISTER which initiates from s and involves $\texttt{read}_\texttt{p}(\texttt{v})$ choose $\texttt{hi}_\texttt{p}$, where $\texttt{hi}_\texttt{p}$ is the number whose existence is implied by the simulation relation R. Steps of WRITE_THROUGH_CACHE involving $\texttt{read\_req}_\texttt{p}()$ and $\texttt{write}_\texttt{p}(\texttt{v})$ actions simulate steps of SEQ_CONSISTENT_REGISTER with the respective actions.

It is straightforward to prove that R satisfies the two conditions of a simulation relation (Definition 2.1). We are not interested in the actual proof, but only in reusing it, i.e., avoiding the need to repeat it.

For the purpose of illustrating proof reuse, we present in Figure 5 a specification of an atomic shared register as a specialization of SEQ_CONSISTENT_REGISTER. The child restricts the allowed values returned by $\texttt{read}_\texttt{p}(\texttt{v})$ to the current value of x by restricting the non-deterministic choice of i to be the index of the latest value in hist-x.

Figure 4 Sequentially consistent shared register specification automaton.

```
AUTOMATON SEQ_CONSISTENT_REGISTER

Signature:                          State:
  Input:    write_p(v)              Seq  hist-x ← ( v_0 )
            read_req_p()            (∀ p ∈ P) last_p ← 1
  Output:   read_p(v)               (∀ p ∈ P) req_p ← 0

Transitions:
INPUT write_p(v)                    OUTPUT read_p(v) choose i
eff: append v to hist-x             pre: req_p > 0
     last_p ← | hist-x |                 v =  hist-x(i)
                                         i ≥ last_p
INPUT read_req_p()                  eff: last_p ← i
eff: req_p ← req_p + 1                   req_p ← req_p - 1
```

Figure 5 Atomic shared register specification.

```
AUTOMATON ATOMIC_REGISTER
          MODIFIES SEQ_CONSISTENT_REGISTER

Transition Restriction:
  OUTPUT read_p(v) choose i
  pre: i = | hist-x |
```

We want to reuse the simulation R to prove that automaton ATOMIC_WRITE_THROUGH_CACHE implements ATOMIC_REGISTER. Since ATOMIC_REGISTER does not extend the states of SEQ_CONSISTENT_REGISTER, the simulation relation does not need to be extended, and it works as is. In general, one may need to extend the parent's simulation relation to capture how the implementation's state relates to the new state added by the specification's child.

To prove that R is also a simulation relation from the child algorithm ATOMIC_WRITE_THROUGH_CACHE to the child specification ATOMIC_REGISTER we have to show two things:

First, we have to show that initial states of ATOMIC_WRITE_THROUGH_CACHE relate to the initial states of ATOMIC_REGISTER. In general, as we prove in Theorem 4.1 below, we need to check the new variables added by the specification child. We need to show that, for any initial state of the implementation, there exists a related assignment of initial values to these new variables. In our example, since ATOMIC_REGISTER does not add any new state, we get this property for free.

Second, we need to show that whenever $\mathtt{R}$ simulates a step of SEQ_CONSISTENT_REGISTER, this step is still a valid transition in ATOMIC_REGISTER. As implied by Theorem 4.1, we only have to check that the new preconditions placed by ATOMIC_REGISTER on transitions of SEQ_CONSISTENT_REGISTER are still satisfied and that the extension of the simulation relation is preserved. Since in our example ATOMIC_REGISTER does not add any new state variables, we only need to show the first condition: whenever $\mathtt{read_p(v)}$ $\mathtt{choose\ i}$ is simulated in ATOMIC_REGISTER, the new precondition "$\mathtt{i} = \mathtt{|hist\text{-}x|}$" holds.

Recall that, when $\mathtt{read_p(v)}$ $\mathtt{choose\ i}$ is simulated in ATOMIC_REGISTER, $\mathtt{i}$ is chosen to be $\mathtt{hi_p}$. For this simulation to work, we need to prove that it is always possible to choose $\mathtt{hi_p}$ to be $\mathtt{|hist\text{-}x|}$. This follows immediately from the added precondition in ATOMIC_WRITE_THROUGH_CACHE, which requires that $\mathtt{read_p(v)}$ only occurs when $\mathtt{synched_p} = \mathtt{true}$, and from the following simple invariant. (This invariant can be proven by straightforward induction.)

Invariant 4.1 *In any reachable state* $\mathtt{t}$ *of* ATOMIC_WRITE_THROUGH_CACHE*:*

$$(\forall\ \mathtt{p} \in \mathtt{P})\quad \mathtt{t.synched_p} = \mathtt{true} \Longrightarrow \mathtt{t.cache_p} = \mathtt{x}$$

Proof Extension Theorem

We now present the theorem which lays the foundation for incremental proof construction. Consider the example illustrated in Figure 1, where a simulation relation $\mathtt{R_p}$ from an algorithm $\mathtt{A}$ to a specification $\mathtt{S}$ is given, and we want to construct a simulation relation $\mathtt{R_c}$ from a specialized version $\mathtt{A}'$ of an automaton $\mathtt{A}$ to a specialized version $\mathtt{S}'$ of a specification automaton $\mathtt{S}$. In Theorem 4.1 we prove that such a relation $\mathtt{R_c}$ can be constructed by supplementing $\mathtt{R_p}$ with a relation $\mathtt{R_n}$ that relates the states of $\mathtt{A}'$ to the state extension introduced by $\mathtt{S}'$. Relation $\mathtt{R_n}$ has to relate every initial state of $\mathtt{A}'$ to some initial state extension of $\mathtt{S}'$ and it has to satisfy a step condition similar to the one in Definition 2.1, but only involving the transition restriction relation of $\mathtt{S}'$.

Theorem 4.1 *Let automaton* $\mathtt{A}'$ *be a child of automaton* $\mathtt{A}$*. Let automaton* $\mathtt{S}'$ *be a child of automaton* $\mathtt{S}$ *such that* $\mathtt{S}' = \mathtt{specialize(S)(N, N_0, TR)}$*. Let relation* $\mathtt{R_p}$ *be a simulation from* $\mathtt{A}$ *to* $\mathtt{S}$*. Let* $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$*.*

A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$*, defined in terms of* $\mathtt{R_p}$ *and* $\mathtt{R_n}$ *as*

$$\{\langle \mathtt{t}, \mathtt{s}\rangle\ :\ \langle \mathtt{t}|_\mathtt{p}, \mathtt{s}|_\mathtt{p}\rangle \in \mathtt{R_p} \wedge \langle \mathtt{t}, \mathtt{s}|_\mathtt{n}\rangle \in \mathtt{R_n}\},$$

is a simulation *from* $\mathtt{A}'$ *to* $\mathtt{S}'$ *if* $\mathtt{R_c}$ *satisfies the following two conditions:*

1. *For any* $\mathtt{t} \in \mathtt{start(A')}$*, there exists a state* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ *such that* $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$*.*
2. *If* $\mathtt{t}$ *is a reachable state of* $\mathtt{A}'$*,* $\mathtt{s}$ *is a reachable state of* $\mathtt{S}'$ *such that* $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p(t|_p)}$ *and* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$*, and* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* $\mathtt{A}'$*, then there exists a finite sequence* α *of alternating states and actions of* $\mathtt{S}'$*, beginning from* $\mathtt{s}$ *and ending at some state* $\mathtt{s}'$*, and satisfying the following conditions:*
   - *(a)* $\alpha|_\mathtt{p}$ *is an execution sequence of* $\mathtt{S}$*.*
   - *(b)* $\forall\ (\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}) \in \alpha$*,* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$*.*
   - *(c)* $\mathtt{s}'|_\mathtt{p} \in \mathtt{R_p(t'|_p)}$*.*
   - *(d)* $\mathtt{s}'|_\mathtt{n} \in \mathtt{R_n(t')}$*.*
   - *(e)* α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t}')$*.*

Proof 4.1: We show that $\mathtt{R_c}$ satisfies the two conditions of Definition 2.1:

1. Consider an initial state $\mathtt{t}$ of $\mathtt{A}'$. By the fact that $\mathtt{R_p}$ is a simulation, there must exist a state $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p(t|_p)}$ such that $\mathtt{s}|_\mathtt{p} \in \mathtt{start(S)}$. By property 1, there must exist a state $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ such that $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$. Consider state $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p}, \mathtt{s}|_\mathtt{n}\rangle$. State $\mathtt{s}$ is in $\mathtt{R_c(t)}$ by definition. Also, $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p}, \mathtt{s}|_\mathtt{n}\rangle \in \mathtt{start(S)} \times \mathtt{N_0} = \mathtt{start(S')}$, where we use the fact that $\mathtt{start(S')} = \mathtt{start(S)} \times \mathtt{N_0}$ (Def. 3.1).
2. First, notice that the assumption on state $\mathtt{s}$ and relation $\mathtt{R_c}$ imply that $\mathtt{s} \in \mathtt{R_c(t)}$ and that properties 2c and 2d imply that $\mathtt{s}' \in \mathtt{R_c(t')}$.

   Next, we show that α is an execution sequence of $\mathtt{S}'$ with the right trace. Indeed, every step of α is consistent with $\mathtt{trans(S)}$ (by 2a) and is consistent with $\mathtt{TR}$ (by 2b). Therefore, by definition of $\mathtt{trans(S')}$ (Def. 3.1), every step of α is consistent with $\mathtt{trans(S')}$. In other words, α is an execution sequence of $\mathtt{S}'$ which starts with state $\mathtt{R_c(t)}$, ends with state $\mathtt{R_c(t')}$ (by 2d), and has the same trace as $(\mathtt{t}, \pi, \mathtt{t}')$ (by 2e).

■

In practice, one would exploit this theorem as follows: The simulation proof between the parent automata already provides a corresponding execution sequence of the parent specification for every step of the parent algorithm. It is typically the case that the same execution sequence, padded with new state variables, corresponds to the same step at the child algorithm. Thus, conditions 2a, 2c, and 2e of Theorem 4.1 hold for this sequence. The only conditions that have to be checked are 2b, and 2d, i.e., that every step of this execution sequence is consistent with the transition restriction $\mathtt{TR}$ placed on $\mathtt{S}$ by $\mathtt{S}'$ and that the values of the new state variables of $\mathtt{S}'$ in the final state of this execution are related to the post-state of the child algorithm.

Note that, we can state a specialized version of Theorem 4.1 for the case of three automata, $\mathtt{A}$, $\mathtt{S}$, and $\mathtt{S}'$, by letting $\mathtt{A}'$ be the same as $\mathtt{A}$. This version would be useful when we know that algorithm $\mathtt{A}$ simulates specification $\mathtt{S}$, and we would like to prove that $\mathtt{A}$ can also simulate a child $\mathtt{S}'$ of $\mathtt{S}$. The statement and the proof of this specialized version are the same as those of Theorem 4.1, except there is no child $\mathtt{A}'$ of $\mathtt{A}$ ($\mathtt{A}' \equiv \mathtt{A}$), so $\mathtt{A}$ must be substituted for $\mathtt{A}'$ and $\mathtt{t}$ for $\mathtt{t}|_{\mathtt{p}}$. In fact, given this specialized version, Theorem 4.1 then follows from it as a corollary because the relation $\{\langle \mathtt{t}, \mathtt{s}\rangle \ : \ \langle \mathtt{t}|_{\mathtt{p}}, \mathtt{s}\rangle \in \mathtt{R_p}\}$ is a simulation relation from $\mathtt{A}'$ to $\mathtt{S}$, and the specialized theorem applies to automata $\mathtt{A}'$, $\mathtt{S}$, and $\mathtt{S}'$.

5 INTERFACE EXTENSION

Interface extension is a formal construct for altering the interface of an automaton and for extending it with new forms of interaction.

For technical reasons, it is convenient to assume that the interface of every automaton contains an empty action ϵ and that its state-transition relation contains empty-transitions: i.e., if $\mathtt{A}$ is an automaton, then

$$(\mathtt{s}, \epsilon, \mathtt{s}') \in \mathtt{trans}(\mathtt{A}) \quad \Leftrightarrow \quad \mathtt{s} = \mathtt{s}'.$$

An interface extension of an automaton is defined using an *interface mapping* function that translates the new (child) interface to the original (parent) interface. New actions added by the child are mapped to the empty action ϵ at the parent. The child's states and start states are the same as those of the parent. The state-transition of the child consists of all the parent's transitions, renamed according to the interface mapping. In particular, the state-transition includes steps that do not change state but involve the new actions (those that map to ϵ).

Definition 5.1 *Automaton* $\mathtt{A}'$ *is an* interface-extension *of an automaton* $\mathtt{A}$ *if* $\mathtt{states}(\mathtt{A}') = \mathtt{states}(\mathtt{A})$, $\mathtt{start}(\mathtt{A}') = \mathtt{start}(\mathtt{A})$, *and if there exists a function* $\mathtt{f}$, *called* interface-mapping[1], *such that*

1. $\mathtt{f}$ *is a function from* $\mathtt{sig}(\mathtt{A}')$ onto $\mathtt{sig}(\mathtt{A})$. *Note that* $\mathtt{f}$ *can map non-*ϵ *actions of* $\mathtt{A}'$ *to* ϵ *(these are the new actions added by* $\mathtt{A}'$*) and is also allowed to be many-to-one.*
2. $\mathtt{f}$ *preserves the classification of actions as "input", "output", and "internal". That is, if* $\pi \in \mathtt{sig}(\mathtt{A}')$, π *is an input action, and* $\mathtt{f}(\pi) \neq \epsilon$, *then* $\mathtt{f}(\pi)$ *is also an input action; likewise, for output and internal actions.*
3. $(\mathtt{s}, \pi, \mathtt{s}') \in \mathtt{trans}(\mathtt{A}') \Leftrightarrow (\mathtt{s}, \mathtt{f}(\pi), \mathtt{s}') \in \mathtt{trans}(\mathtt{A})$.

Notation 5.2 *Let* $\mathtt{A}'$ *be an interface-extension of* $\mathtt{A}$ *with an interface-mapping* $\mathtt{f}$.

If α *is an execution sequence of* $\mathtt{A}'$, *then* $\alpha|^{\mathtt{f}}$ *denotes a sequence obtained by replacing each action* π *in* $\mathtt{alpha}$ *with* $\mathtt{f}(\pi)$, *and then collapsing every transition of the form* $(\mathtt{s}, \epsilon, \mathtt{s})$ *to* $\mathtt{s}$.

Likewise, if β *is a trace of* $\mathtt{A}'$, *then* $\beta|^{\mathtt{f}}$ *denotes a sequence obtained by replacing each action* π *in* β *to* $\mathtt{f}(\pi)$, *and by subsequently removing all the occurrences of* ϵ.

The following theorem formalizes the intuition that the sets of executions and traces of an interface-extended automaton are equivalent to the respective sets of the parent automaton, modulo the interface-mapping. The proof is straightforward by induction using Definition 5.1 and Notation 5.2.

Theorem 5.1 *Let automaton* $\mathtt{A}'$ *be an interface extension of* $\mathtt{A}$ *with an interface-mapping* $\mathtt{f}$.

Let α *be a sequence of alternating states and actions of* $\mathtt{A}'$ *and let* β *be a sequence of external actions of* $\mathtt{A}'$. *Then:*

1. $\alpha \in \mathtt{execs}(\mathtt{A}') \Leftrightarrow \alpha|^{\mathtt{f}} \in \mathtt{execs}(\mathtt{A})$.
2. $\beta \in \mathtt{traces}(\mathtt{A}') \Leftrightarrow \beta|^{\mathtt{f}} \in \mathtt{traces}(\mathtt{A})$.

When interface extension is followed by the specialization modification, the resulting combination corresponds to the notion of modification by *subclassing for extension* [8]. The resulting child specializes the parent's behavior and introduces new functionality. Specifically, a specialization of an interface-extended automaton may add transitions involving new state components and new interface. The generalized definition of the parent-child relationship is then as follows:

[1] Interface-mapping is similar to *strong correspondence* of [38].

Definition 5.3 *Automaton* $\mathtt{A}'$ *is a* child *of an automaton* $\mathtt{A}$ *if* $\mathtt{A}'$ *is a specialization of an interface extension of* $\mathtt{A}$.

Theorem 5.1 enables the use of the proof extension theorem (Theorem 4.1) for this parent-child definition, once the child's actions are translated to the parent's actions using the interface mapping of Definition 5.1.

6 PRACTICAL EXPERIENCE WITH INCREMENTAL PROOFS

In this section we describe our experience designing and modeling a complex group communications service (see [26]), and how the framework presented in this paper was exploited. We then describe an interesting modeling methodology that has evolved with our experience in this project.

Group communication systems (GCSs) [3, 37] are powerful building blocks that facilitate the development of fault-tolerant distributed applications. GCSs typically provide reliable multicast and group membership services. The task of the *membership service* is to maintain a listing of the currently active and connected processes and to deliver this information to the application whenever it changes. The output of the membership service is called a *view*. The reliable multicast services deliver messages to the current view members.

Traditionally, GCS developers have concentrated primarily on making their systems useful for real-world distributed applications such as data replication (e.g., [16]), highly available servers (e.g., [5]) and collaborative computing (e.g., [7]). Formal specifications and correctness proofs were seldom provided. Many suggested specifications were complicated and difficult to understand, and some were shown to be ambiguous in [4]. Only recently, the challenging task of specifying the semantics and services of GCSs has become an active research area.

The I/O automaton formalism has been recently exploited for specifying and reasoning about GCSs (e.g., in [9, 11, 12, 15, 24, 28]). However, all of these suggested I/O automaton-style specifications of GCSs used a single abstract automaton to represent multiple properties of the same system component and presented a single algorithm automaton that implements all of these properties. Thus, no means were provided for reasoning about a subset of the properties, and it was often difficult to follow which part of the algorithm implements which part of the specification. Each of these papers dealt with proving correctness of an individual service layer and not with a full-fledged system.

In [26], we modeled a full-fledged example spanning the entire virtually synchronous reliable group multicast service. We provided specifications, formal algorithm descriptions corresponding to our actual C++ implementation, and also simulation proofs from the algorithms to the specifications. We employed a client-server approach: We presented a virtually synchronous group multicast client that interacts with an external membership server. Our virtually synchronous group multicast client was implemented using approximately 6000 lines of C++ code. The server [27] was developed by another development team also using roughly 6000 lines of C++ code. Our group multicast service also exploits a reliable multicast engine which was implemented by a third team [34] using 2500 lines of C++ code.

We sought to model the new group multicast service in a manner that would match the actual implementation on one hand, and would allow us to verify that the algorithms meet their specifications on the other hand. In order to manage the complexity of the project at hand we found a need for employing an object-oriented approach that would allow for reuse of models and proofs, and would also correspond to the implementation, which in turn, would reuse code and data structures.

In [26], we used the I/O automaton formalism with the inheritance-based incremental modification constructs presented in this paper to specify the safety properties of our group communication service. We specified four abstract specification automata which capture different GCS properties: We began by specifying a simple GCS that provides reliable FIFO multicast within views. We next used the new inheritance-based modification construct to specialize the specification to require also that processes moving together from one view to another deliver the same set of messages in the former. We then specialized the specification again to also capture the Self Delivery property which requires processes to deliver their own messages. The fourth automaton specified a stand-alone property (without inheritance) which augments each view delivery with special information called *transitional set* [37].

We then proceeded to formalize the algorithms implementing these specifications. We first presented an algorithm for within-view reliable FIFO multicast and provided a five page long formal simulation proof showing that the algorithm implements the first specification. Next, we presented a second algorithm as an extension and a specialization of the first one. In the second algorithm, we restricted the parent's behavior according to the second specification, i.e., we added the restriction that processes moving together from one view to another deliver the same set of messages in the former. Additionally, in the second algorithm, we extended the service interface to convey transitional sets, and added the new functionality for providing clients with transitional sets as per the fourth specification. By exploiting

Theorem 4.1, we were able to prove that the second algorithm implements the second specification (and therefore also the first one) in under two pages without needing to repeat the arguments made in the previous five page proof. We separately proved that the algorithm meets the fourth specification. Finally, we extended and specialized the second algorithm to support the third property. Again, we exploited Theorem 4.1 in order to prove that the final algorithm meets the third specification (and hence all four specifications) in a merely two and a half page long proof.

We are currently continuing our work on group communication. We are incrementally extending the system described in [26] with new services and semantics using the same techniques.

A Modeling Methodology

Specialization does not allow children to introduce behaviors that are not permitted by their parents and does not allow them to change state variables of their parents. However, when we modeled the algorithms in [26], in one case we saw the need for a child algorithm to modify a parent's variable. We dealt with this case by introducing a certain level of non-determinism at the parent, thereby allowing the child to resolve (specialize) this nondeterminism later.

In particular, the algorithm that implemented the second specification described above sometimes needed to forward messages to other processes, although such forwarding was not needed at the parent. The forwarded messages would have to be stored at the same buffers as other messages. However, these message buffers were variables of the parent, so the child was not allowed to modify them. We solved this problem by adding a forwarding action which would forward arbitrary messages to the parent automaton; the parent stored the forwarded messages in the appropriate message buffers. The child then restricted this arbitrary message forwarding according to its algorithm.

We liken this methodology to the use of *abstract methods* or *pure virtual methods* in object-oriented methodology, since the non-determinism is left at the parent as a "hook" for prospective children to specify any forwarding policy they might need. In our experience, using this methodology did not make the proofs more complicated.

7 DISCUSSION

We described a formal approach to incrementally defining specifications and algorithms, and incorporated an inheritance-based methodology for incrementally constructing simulation proofs between algorithms and specifications. This technique eliminates the need to repeat arguments about the original system while proving correctness of a new system.

We have successfully used our methodology in specifying and proving correct a complex group communication service [26]. We are planning to experiment with our methodology in order to prove other complex systems.

We have presented the technique mathematically, in terms of I/O automata. Furthermore, the formalism presented in this paper and the syntax of incremental modification is consistent with the continued evolution of the IOA programming and modeling language. Since IOA is being developed as a practical programming framework for distributed systems, one of our goals is to incorporate our inheritance-based modification technique and approach to proof reuse into the IOA programming language toolset [17, 18].

Future plans also include extending our proof-reuse methodology to a construct that allows a child to modify the state variables of its parent. Other future plans include adding the ability to deal with multiple inheritance. In all of our work, we aim to formulate and extend formal specification techniques that would be useful for practical software development.

ACKNOWLEDGMENTS

We thank Paul Attie, Steve Garland, Victor Luchangco and Jens Palsberg for their helpful comments and suggestions.

REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] ACM. *Commun. ACM 39(4), special issue on Group Communications Systems*, April 1996.

[4] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Comp. Sci. TR 95-1534, Cornell Univ., Aug. 1995.

[5] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. *19th Intern. Conference on Distr. Computing Systems (ICDCS)*, pp. 244–252, June 1999.

[6] M. Bickford and J. Hickey. An object-oriented approach to verifying group communication systems. http://www.cs.cornell.edu/jyh/papers/cav99_ooioa/.

[7] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. *Multimedia Computing and Networking (MMCN98)*, 1998.

[8] T. Budd. *An Introduction to Object-Oriented Programming, 2nd Edition*. Addison Wesley Longman, 1996.

[9] G. V. Chockler. An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[10] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Also OOPSLA'89.

[11] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 227–236, June 1998.

[12] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic primary configuration group communication service. *13th International Symposium on DIStributed Computing (DISC)*, pp. 64–78, 1999.

[13] W. P. de Roever and K. Engelhardt. *Data Refinement Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, Dec. 1998.

[14] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science, special issue on Distributed Algorithms*, 220, 1999.

[15] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 53–62, August 1997.

[16] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. *16th IEEE Intern. Symp. on Reliable Distrib. Systems*, October 1997.

[17] S. J. Garland and N. A. Lynch. *Foundations of Component Based Systems*, chapter Using I/O Automata for Developing Distributed Systems. Cambridge University Press, USA, 1999. To appear.

[18] S. J. Garland, N. A. Lynch, and M. Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems*. MIT LCS, Dec. 1997. http://sds.lcs.mit.edu/∼garland/ioaLanguage.html.

[19] M. Hayden and R. van Renesse. Optimizing Layered Communication Protocols. TR96-1613, Dept. of Computer Science, Cornell University, November 1996.

[20] M. P. Heimdahl and C. L. Heitmeyer. Formal methods for developing high assurance computer systems: Working group report. *Second IEEE Workshop on Industrial-Strength Formal Techniques*, Oct. 1998.

[21] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. *Real Time Systems Symposium*, Dec. 1994. Full version: MR-7619, Naval Research Laboratory.

[22] C. L. Heitmeyer. On the need for 'practical' formal methods. *Formal Techniques in Real-Time Fault-Tolerant Systems. 5th Intern. Symposium*, pp. 18–26, Sept. 1998. LNCS 1486 (invited paper).

[23] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. T. Ito and A. R. Meyer, editors, *Proceedings of Theoretical Aspects of Computer Software*, pp. 548–568. LNCS 526, 1991.

[24] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, Mar. 1999.

[25] S. Kamin. Inheritance in Smalltalk–80: A denotational definition. *15th Symp. on Principles of Programming Languages*, pp. 80–87, 1988.

[26] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: Specifications, algorithms and proofs. TR 794, MIT Lab. for Comp. Science, Nov. 1999. To appear in ICDCS 2000. http://theory.lcs.mit.edu/∼idish/Abstracts/vs.html.

[27] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. TR CS99-623, Comp. Sci., Univ. of California, San Diego, June 1999.

[28] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. *12th International Symposium on DIStributed Computing (DISC)*, pp. 258–272, Sept. 1998.

[29] B. Lampson. Generalizing Abstraction Functions. MIT, Laboratory for Computer Science, Principles of Computer Systems, Handout 8, 1997. ftp://theory.lcs.mit .edu/pub/classes/6.826/www/6.826-top.html.

[30] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[31] N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *27th IEEE Fault-Tolerant Computing Symposium (FTCS)*, pp. 272–281, 1997.

[32] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quart.*, 2(3):219–246, '89.

[33] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. *Proc. of ACM Conference on Lisp and Functional Programming*, pp. 289–297, 1988.

[34] I. Shnaiderman. Implementation of Reliable Datagram Service in the LAN environment. Lab project, The Hebrew University of Jerusalem, January 1999. http://www.cs.huji.ac.il/∼transis/publications.html.

[35] A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39(1):45–49, July 1991.

[36] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. *10th Conf. on Object-Oriented Progrgamming Systems, Lang., and Appl. (OOPSLA)*, vol. 30 of *ACM SIGPLAN*, pp. 200–214, Oct. 1995.

[37] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. TR CS99-31, Institute of Comp. Science, The Hebrew University of Jerusalem, Israel, Sept. 1999.

[38] D. Yates, N. Lynch, V. Luchangco, and M. Seltzer. I/O automaton model of operating system primitives. Master's thesis, Harvard University and MIT, May 1999.