Modelling Techniques for Evolving Distributed Applications

R. Sekar, Y.-J. Lin and C.R. Ramakrishnan

Abstract

Several languages and techniques have been proposed for formal specification and validation of concurrent systems. However, these techniques provide no support for modelling incremental changes that take place during software development, such as successive refinements that take place during the design phase or changes that take place later on as a result of software evolution. Consequently any changes to the system model need to be incorporated by manual editing of the system specification, which is cumbersome and error-prone. Moreover, editing being an uncontrolled process, there is no way to automatically carry over (most of the) correctness properties after minor changes to the system. These factors can make formal approaches very expensive for large and evolving systems. To alleviate this problem, we present a language RL[1] in this paper that provides syntactic as well as semantic support for modelling incremental changes. Based on the language mechanisms, we then present a method for automatically carrying over properties after refinement. We also present algorithms for compiling RL specifications into finite state automata (FSA) that can be analyzed using traditional algorithms for establishing new properties that hold only after refinement.

Keywords: D.2.1, D.2.4/F.3.1, D.3.3, D.1.3, D.2.2

1 Introduction

Distributed systems are increasingly being used to accomplish complex and critical tasks. However, as compared to centralized systems, distributed systems are inherently harder to program, debug or maintain. Formal modelling and reasoning techinques are therefore being employed to simplify

[1] RL stands for **R**efinement **L**anguage.

these problems. CSP [9], Estelle [4], Lotos [3], SDL [2] and Petri nets [13] are among many such formal languages. A variety of tools for programming and/or reasoning about specifications in these languages (or variants of these languages) are currently available (e.g., SMV[5], SPIN[10] and COSPAN[11]) and have been used with a good degree of success.

We have been using formal techniques for modelling and reasoning about telecommunication software systems, which are among the largest distributed systems that exist today. Such large systems go through many incremental changes over their life cycle, such as successive refinements that take place during the design phase or changes that take place later on as a result of software evolution. Unfortunately, the techniques mentioned above do not provide support for modelling such changes. Consequently any changes to the system model needs to be incorporated by manual editing of the system specification. This editing process is cumbersome and error-prone: in the absence of appropriate language support, even conceptually small changes can require editing of the entire system specification and making changes in a number of places. Moreover, since editing is an uncontrolled process, all correctness properties need to be reestablished from scratch, after even minor changes to the specification. These factors can make model-based formal approaches very expensive for large and evolving distributed systems such as telecommunication systems. In order to alleviate these problems, we have developed a new modelling language RL that provides syntactic as well as semantic support for expressing incremental changes. Below, we provide an overview of the language, followed by a summary of results.

1.1 Overview of RL

Our overall approach to modelling is similar to CSP [9], in that a system consists of a set of sequential processes that communicate through synchronous channels. The behavior of a process is described by a parameterized module that can be instantiated. A module, in turn, is characterized by a set of state variables and (non-deterministic) transition rules. An example of a simple module specification appears in Figure 1. It gives a simple (incomplete) specification of a telephone that is characterized by a state variable that describes the on-hook or off-hook status of the phone, and two transition rules that correspond to lifting and replacing the handset.

In order to support evolution, we need to model changes that alter the behavior of a module. These changes can take place through modifications to the state component and/or transition rules. In the former case, the

```
type HookStatus = {idle, offhook};
type Usr2Phone = {liftHandSet, replaceHandSet};

module phone1(hookState, usr2phone) {
    var hookState: HookStatus;
    chan usr2phone: Usr2Phone;

    initial hookState := idle;

    trans
        usr2phone?liftHandSet, hookState = idle   → hookState := offhook;
        usr2phone?replaceHandSet, hookState = offhook   → hookState := idle;
}
```

Figure 1: A Simple Module Specification in RL

change can be addition of new state variables, or expansion of the domain of values assumed by existing state variables. In the latter case, the change can be addition or deletion of new transition rules, or selective modifications to existing rules by strengthening or relaxation of the pre and post conditions. These changes can be conveniently described in RL using the following mechanisms.

Module Derivation: All changes are expressed through module derivation, a concept similar to class derivation in object-oriented (OO) languages. By specifying priority relationships among rules in the derived and base modules, the rules in a derived module can override or selectively modify the rules in the base modules. New state variables can also be specified as part of a derived module. More importantly, we can alter the domain of an existing state variable in a structured manner using the following *type derivation* mechanism.

Type Derivation: New types can be derived from existing types either by *refining* a single value into multiple values, or by *augmenting* the type with entirely new values. Even when an existing state variable is redefined to possess a derived type in a derived module, most of the original rules that operate on the base data type can be reused in the derived module. Moreover, the type derivation mechanism serves as the basis of a mapping between the derived and original module specifications that is used in *incremental validation*.

In comparison with OO-languages, modules are analogous to classes, with message transmission taking the role of member function invocation, and (sets of) transition rules taking the role of member function bodies. Addition of new state variables corresponds to having additional data members in the derived class, and addition of new transition rules is similar to adding new member functions. However, for the purposes of supporting refinement and evolution, RL improves upon traditional OO-languages. First, the notion of type derivation has no direct analog in OO-languages[2]. Second, priorities provide a more flexible mechanism for refinement than the override mechanism in OO-languages, where a new member function always overrides a function in the base class with the same name.

With regards to our type derivation mechanism, the notion of refinement was first developed in the context of statecharts [8]. Statecharts is a visual formalism, where "boxes" are used to denote states of a machine. In this context, refinement was viewed as a way to decompose a single (super) state into multiple substates. In the context of text-based, typed language such as RL, we needed to develop a different view of refinement. Specifically, we view refinement as operating upon a data type, rather than a state. This view requires us to ascribe much more structure and semantics to refinement, as compared to statecharts. For instance, we need to extend refinement to structured data types, as well as define the semantics of comparisons and assignments over these types. Our notion of type augmentation was not considered in the context of statecharts.

1.2 Summary of Results

Several issues arise in the design of a language for modelling refinement and evolution. In this section, we summarize these issues and outline how we resolve them.

- Design of mechanisms for expressing incremental changes: the design must minimize the potential conflict between expressive power of the mechanism and simplicity of semantics. The former aspect determines the ease with which we can model evolution. The latter aspect defines the ease of understanding or reasoning about incremental modifications. The module and type derivation mechanisms, described in Section 2, have been designed to balance these considerations. We illustrate the use of these mechanisms through examples in Section 3.

[2]Type derivation may appear to be similar to inheritance, since functions operating on the base class continue to operate on the derived class in OO-languages. However the important distinction here is that the semantics of rules that were written for a type continue to remain applicable *even at the new values* added by the derivation process, whereas in OO-languages the functions on the base class can operate only on components of the base type.

- Development of a methodology for incremental validation: In Section 4, we present our approach for *automatically* carrying over properties after incremental modifications.

- Enable reuse of existing techniques and tools: When we cannot use incremental validation, e.g., for properties that do not hold before the refinement, we need to "validate from scratch." Several efficient techniques have been developed for such validation of FSAs. In order to reuse these techniques, we have develop algorithms in Section 5 for compiling RL specifications into FSAs. The compilation "translates away" type and module derivation mechanisms *without* undue increase in either the state-space or the complexity of transition rules in the resultant automaton.

- Collapsing sequences of incremental modifications: While the language mechanisms to express incremental changes can make it easier to model and validate evolving software systems, they have the disadvantage that after several modification steps, it becomes difficult to understand the specification. Our semantics of module derivation, as well as the compilation algorithms, enable us to collapse a module description that involves a sequence of module derivation steps into a single module description that is much more understandable.

2 Overview of RL

A specification in RL consists of global declarations and module declarations, as shown in the example in Figure 1. A module is characterized by a set of module parameters, declarations, priority specifications and transition rules. Module parameters are analogous to (formal) parameters to a procedure. They are substituted by actual parameters at the time of *module instantiation*, as described later on in the section. Of the three kinds of declarations, type declarations associate names with *type expressions*; variable and channel declarations are used to declare variables and channels respectively.

Transition rules are of the form $Guard \rightarrow Effect$, where $Guard$ specifies the conditions under which the rule can be executed, and can include tests for presence of messages on channels or comparisons involving state variables. $Effect$ includes output operations on channels or assignments to variables. A rule is executable if all the conditions in its guard are satisfied. Rule execution is *atomic*, i.e., either all the assignments in its body are executed or no assignment is executed.

Channels in RL are not buffered, and thus provide a rendezvous mechanism for communication. This enables us, in the manner described below, to view channel inputs and outputs as merely syntactic sugar for a particular way of using the state variables. We view each channel as a state variable that can take a value belonging to the channel's type, or be ϵ that denotes an empty channel. Output on a channel is treated as a combination of a test to ensure that the channel is empty, followed by an assignment of the value to be output on the channel. Similarly, an input operation $chan?expr$ is equivalent to the comparison $chan = expr$, followed by the assignment $chan := \epsilon$. We remark that an input operation is viewed in our language as a comparison, as opposed to the more traditional view of an assignment. This is because the effect of an assignment can be achieved using the *pattern-matching* mechanism described below.

A pattern-matching mechanism is supported for equality comparisons through the use of *temporary variables*. The scope of such variables is limited to a single transition rule. When used in an equality comparison, they get *bound* to a value that satisfies the comparison. Subsequent uses of the variable refer to this value. Pattern-matching provides a convenient way to deal with structured data types. For instance,

$$\mathsf{in}?f(\mathsf{X},\ \mathsf{Y}) \;\rightarrow\; \mathsf{z1} := \mathsf{X},\ \mathsf{z2} := \mathsf{Y}$$

tests for the presence of a message with root symbol f on the channel in, and if present, assigns the children of f to z1 and z2 respectively. As in the above example, we use the convention that temporary variable names start with a capital letter, whereas state variable names begin in lower case. In conjunction with module instantiation, pattern-matching provides a mechanism for supporting polymorphism.

2.1 Type Derivation

Type derivation permits us to derive new data types from existing ones. It provides a structured way to extend existing specifications so that existing rules (operating on base types) can be reused whenever they are meaningful in the context of the derived type. The type derivation mechanism is also designed so that properties established for existing specifications can be carried over to refined version of the specification (see Section 4).

New types can be derived from existing types, which can in turn be derived from other types, or be *base types*. The base types in the language include integer subranges (e.g., type $\mathsf{t1}$ below), enumerated lists of flat (type $\mathsf{PhoneNum}$) or structured (type $\mathsf{t2}$) values, as well as arrays (type $\mathsf{t3}$) over these types:

$$\textbf{type } \mathsf{t1} = 1..20$$
$$\textbf{type } \mathsf{PhoneNum} = \{p1, p2, p3\}$$
$$\textbf{type } \mathsf{t2} = \{\ c, d(\mathsf{t1}), f(\mathsf{HookStatus}, \mathsf{PhoneNum})\ \}$$
$$\textbf{type } \mathsf{t3} = \textbf{array } [1..10] \textbf{ of } \mathsf{PhoneNum}$$

Declaration of $\mathsf{t2}$ refers to the $\mathsf{HookStatus}$ type in Figure 1. In type $\mathsf{t2}$, d is a constructor of arity 1, and f has arity 2. Examples of terms in $\mathsf{t2}$ include $c, d(3)$ and $f(idle, p2)$. We note that construction of recursive types is not permitted in the language.

Type derivation in RL is provided by *refinement* and *augmentation*. Refinement allows decomposition (or specialization) of a value into multiple values, whereas augmentation allows us to add a new value to a type. For instance, we can define a new type $\mathsf{HookStatus1}$ by specializing the value $offhook$ in $\mathsf{HookStatus}$ of Figure 1 as follows:

$$\textbf{type } \mathsf{HookStatus1} = \textbf{refine } \mathsf{HookStatus} \textbf{ at } offhook \textbf{ into } \{connected, dialtone\};$$

Whenever any of these refined values appears in a context where the $\mathsf{HookStatus}$ type is expected, it is treated as if the value $offhook$ appeared there. This provides the mechanism for reuse of rules that were designed to operate on $\mathsf{HookStatus}$. For structured values, the refinement declaration specifies a refinement for the root as well as the children:

$$\textbf{type } \mathsf{t4} = \textbf{refine } \mathsf{t2} \textbf{ at } f(X, Y) \textbf{ into } \{g(X, Y, \mathsf{t1}), h(Y, X)\};$$
$$\textbf{type } \mathsf{t5} = \textbf{refine } \mathsf{t2} \textbf{ at } f(X : \mathsf{HookStatus}, Y) \textbf{ into } \{f(X : \mathsf{HookStatus1}, Y)\};$$

The first declaration specifies that any value of the form $g(X, Y, n)$ or $h(Y, X)$, (where X, Y and n are any values in types $\mathsf{HookStatus}$, $\mathsf{PhoneNum}$ and $\mathsf{t1}$ respectively) is a refinement of $f(X, Y)$. In the second declaration, the refinement mapping on the type of a child introduces a natural mapping on the type of the parent. A refined type includes all the terms in the base type excluding those terms that were refined, plus the new terms added by the refinement. Thus $\mathsf{HookStatus1}$ contains $idle, connected$ and $dialtone$.

While type refinement permits us to specialize values, type augmentation provides a mechanism for defining supertypes of existing types by adding entirely new values. The nature of augmentation permits full reuse of rules defined on the base types.

$$\textbf{type } \mathsf{HookStatus2} = \textbf{augment } \mathsf{HookStatus1} \textbf{ by } \{ring\};$$

Type augmentation can be used in conjunction with refinement to add new values in a structured manner. In particular, if a type T' is derived from another type T, then new values can be added to T' as a refinement of an existing value in T:

type $\mathsf{HookStatus3}$ = **augment** $\mathsf{HookStatus2}$ **at** $offhook$ **by** $\{wait(\mathsf{PhoneNum})\}$;

2.1.1 Semantics of Type Derivation

The elements in the types HookStatus through HookStatus3 are shown in Figure 2, where additional elements (of the form $\perp$) have been added to simplify the semantics of type augmentation. The values added in this manner are called $\perp$-values, whereas the other values are said to be *proper*. Type augmentation can now be viewed as refinement at one of the $\perp$-values. Augmentations at the outermost level (as in HookStatus2) can be understood as refinement of the undefined element $\perp$ that is added to every enumerated type. Similarly, augmentation at some value v with new values $v_1, ..., v_n$ has the meaning that $v_1, ..., v_n$ are to be treated as refinements of v. To get this effect, whenever v is refined, we add a new element $\perp_v$ as one of its specializations (see $\perp_{offhook}$ in Figure 2), and later on, refine $\perp_v$ into $v_1, ..., v_n$ to get the effect of augmentation. Thus the $\perp$-elements act as "place-holders" where additional elements can be added in derived types.

With the above view of augmentation as an instance of refinement, we can understand type derivation in terms of the structure of resultant types. Figure 2 shows the associations among the types derived from HookStatus. In particular, a type derivation operation on a type T to get another type $\mathsf{T'}$ defines a (many-to-one) mapping from values in type $\mathsf{T'}$ to values in type T. We use the notation $\mathcal{M}_{\mathsf{T'}\to\mathsf{T}}$ to denote this mapping. For instance,

$$\mathcal{M}_{\mathsf{HookStatus3}\to\mathsf{HookStatus}}(dialtone) = offhook$$

When a value v in T is refined into $v_1, ..., v_n$ in type $\mathsf{T1}$, $\mathcal{M}_{\mathsf{T1}\to\mathsf{T}}$ maps $v_1, ..., v_n$ to v and other values to themselves. We use the notation $\mathcal{M}_{\mathsf{T}}(val)$ to denote the value mapped onto by val in the type T. In order for this mapping to be well-defined, we require that the value val belong to exactly one type derived from T. If val does not belong to T or any of its derived types, we set $\mathcal{M}_{\mathsf{T}}(val) = \perp$.

Based on the semantics of type derivation, we now specify the semantics of assignments and comparisons. An assignment of the form $\mathsf{v} := \mathsf{Expr}$ is valid iff the type $\mathsf{T'}$ of Expr is the same as the type T of the variable v, or is derived from it. Moreover, $\mathcal{M}_{\mathsf{T'}\to\mathsf{T}}(\mathsf{Expr})$ must be a proper element of T. The meaning of the assignment is to first evaluate Expr to yield a value V and then assign $\mathcal{M}_{\mathsf{T'}\to\mathsf{T}}(V)$ to the variable v. Since a value in a derived type has a unique image in the base type, we can permit the expression to have a type that is refined from that of the variable, but not vice-versa.

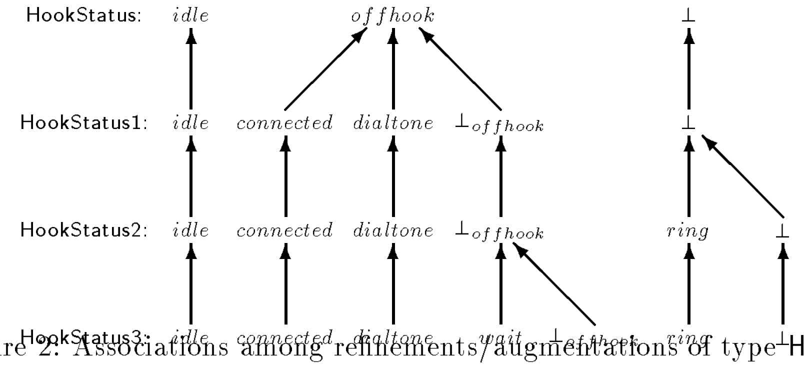

Figure 2: Associations among refinements/augmentations of type HookStatus

All comparisons are of the form *variable op expr*, where *op* can be one of {`=, !=, <, <=, >, >=`} for integer variables, and one of {`=, !=`} for others. The semantics of equality is defined as follows. A comparison $\mathsf{v} = \mathsf{Expr}$ is permissible whenever the types T' and T'' of the operands have a least common ancestor type T such that

$$\mathcal{M}_{\mathsf{T}' \to \mathsf{T}}(\mathsf{v}) = \mathcal{M}_{\mathsf{T}'' \to \mathsf{T}}(\mathsf{Expr}) = \text{some proper value in type } \mathsf{T}$$

For inequality, we require that both operands map to *different proper values* in T.

2.2 Module Derivation

Module instantiation, expressed as a declaration of a module type variable, provides the mechanism for module composition and derivation. Module composition is used to synthesize new modules by *composing* existing modules, whereas module derivation is used to incrementally modify the behavior of a module. The distinction between composition and derivation is in the specification writer's view, but not in the language.

The behavior of an existing module can be modified in a derived module either by modifying the state, or the transition rules of the base module. In the former case, it can either be addition of new state variables or enlargement of the domain of an existing variable through type derivation. In either case, new transition rules are typically specified as part of the derived module that deal with the change in the domain. Priority relationships can be specified between the rules in the base module and those in the derived module. Priorities have the obvious semantics that a lower priority rule (or initialization) is *never* executed when a higher priority rule is executable.

Use of module derivation is illustrated through examples in the Section 3. Incremental validation is discussed in Section 4.

The declaration of a variable

var m1: **module** m(a1,...,aN);

within a module m2 creates an instance of the module m. Any variable v inside the module instance m1 can be accessed within m2 using the notation m1.v. In simple terms, this instantiation is equivalent to importing the declarations and the transition rules of m into m2, after substituting the *formal parameters* that appear in the definition of m with the *actual parameters* a1,...,aN. For the substitution to succeed, the types of actual and formal parameters must be compatible. In particular, if the actual parameter is a variable then the type of the formal parameter must be derived from that of the actual parameter or vice-versa. The type of the parameter in the instantiated module is taken to be the more-refined of the formal and actual parameter types. The idea here is that the values in a derived type form a superset of the values in the base type. Hence, a variable of derived type can be thought of as *simultaneously* holding the values in the derived type as well as the base type. This view permits us to reuse most of the transition rules in m that may have been specified in the context of a less-refined type.

2.3 Semantics of RL Specifications

The state of a system specified in RL is given by a function S that maps variables in the specification to values. The behavior of the specification is defined as follows, based on the relation $\mathbf{T}$ that specifies when the system can transition from one state to another.

Definition 1 (History and Behavior) *A* history *is a (potentially infinite) sequence of states $S_0, S_1, \ldots$ such that S_0 is the initial state given by the specification and for any two successive states S_i and S_{i+1}, the condition $\mathbf{T}(S_i, S_{i+1})$ holds. The* behavior *of an RL specification is the set of all histories of the specification.*

Validation of RL specifications is based on finite representations of behavior. On the other hand, simulation of RL programs results in the enumeration of one of the histories.

Due to space limitations, we only provide a brief description of the transition relation $\mathbf{T}$. It is determined by (a) the semantics of rule execution, and (b) the method for identifying a rule that can executed in a given state. To

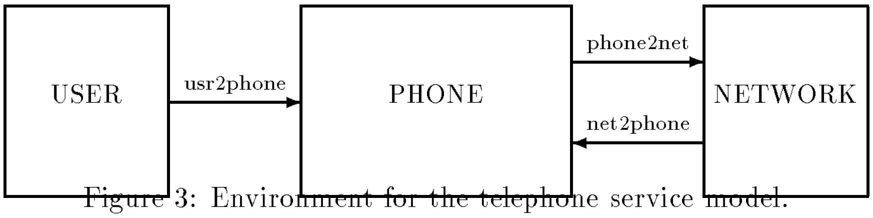

Figure 3: Environment for the telephone service model.

define rule execution semantics, we need to specify the semantics of expressions, comparisons and assignments, which are all based on the $\mathcal{M}$ function described in Section 2.1.1. For a rule $C \rightarrow A$, we say that the rule is *enabled* in state S if all the comparisons in C hold, with some bindings B for temporary variables. If the rule is enabled, the semantics of its execution is given by the new state S' obtained from S by performing all the assignments in A in sequence, with the temporary variable bindings as in B.

To define when a rule is executable, we can define a *select* function that searches through the rules in a specification and returns all those that are executable. The search starts in the main module and proceeds recursively through the modules that are instantiated. The search proceeds from highest priority rules to the lowest such that (i) once a higher priority rule is enabled, lower priority rules are eliminated from consideration, and (ii) when the search proceeds into an instantiated module, appropriate parameter passing (by reference) is taken into account.

3 Illustration of RL

In this section, we present an example of a telephone service to demonstrate the convenience of the language mechanisms in RL for modelling refinement and evolution. Although we expect these mechanisms to be used mainly on large systems to model incremental changes, for the sake of simplicity of illustration, we use only a small example. The environment for our example is shown in Figure 3. The USER module models actions of a telephone user, whereas NETWORK models the rest of the telephone network. We will not be specifying these two modules, but only the PHONE module. We start with a simple specification that deals only with the interaction between the telephone service and the user of the service. In this simplified view, the phone is either idle (on hook) or off the hook, with the lift-hand-set and replace-hand-set messages from the user controlling the current state of the device. The simplified model is shown in Figure 1.

As the next step of the modelling process, we model the distinction between lifting a hand set to accept an incoming call and lifting it to make an outgoing call. This is done by refining the HookStatus type as shown below. We also need to introduce a new state corresponding to the state when the phone is ringing.

type HookStatus1 = **refine** HookStatus **at** $offhook$ **into** $\{connected, dialtone\}$;
type HookStatus2 = **augment** HookStatus1 **by** $\{ring\}$;

Along with these type declarations, we introduce rules for transitions between the new states. At this point, we do not wish to exactly specify the conditions under which hookState can change from *dialtone* to *connected*, or toggle between *idle* and *ring*. However, since we are developing an abstract model of the system, we would like to capture all possible behaviors of the system and so we specify the possibility of making transitions between these states on receipt of some (unspecified) messages from the network. There is also the possibility that some messages may be received in inappropriate states, so that it is possible to simply discard them. (Otherwise, such messages can fill up the message buffers and prevent further messages from being received.) The derived module description is shown below:

```
type FromNetwork = {signal};
module phone2(hookState, usr2phone, net2phone) {
    var hookState: hookStatus2;
    var phone: module phone1(hookState, usr2phone);
    chan net2phone: FromNetwork;
    priority phone < phone2;
    trans
        usr2phone?liftHandSet, hookState = idle  → hookState := dialtone;
        usr2phone?liftHandSet, hookState = ring  → hookState := connected;
        net2phone?signal, hookState = dialtone  → hookState := connected;
        net2phone?signal, hookState = idle  → hookState := ring;
        net2phone?signal, hookState = ring  → hookState := idle;
        net2phone?signal  → ;
}
```

The declaration of the variable phone causes instantiation of the module phone1. Because the module phone2 specifies that rules in phone2 have a higher priority, the first rule in phone1 will never be applicable. However, the second rule (as well as the initialization) continues to be applicable in the two states that correspond to $offhook$.

As a next step, we incorporate interactions corresponding to outgoing calls. To do this, we add the *dial* message to usr2phone, and introduce two new states under $offhook$:

type Usr2Phone1 = **augment** Usr2Phone **by** $\{dial($PhoneNum$)\}$;
type HookStatus3 = **augment** HookStatus2 **at** offhook **by** $\{wait, busy\}$;

We also refine the message from network to phone, and introduce a channel for communication from the phone to the network. The revised specification is:

type ToNetwork = $\{call($PhoneNumber$)\}$;
type FromNetwork1 = **refine** FromNetwork **at** $signal$ **into** $\{accept, busy\}$;
module phone3(hookState, usr2phone, net2phone, phone2net) {
  chan usr2phone: Usr2Phone1;
  chan phone2net: ToNetwork; net2phone: FromNetwork1;
  var hookState: HookStatus3;
  var phone: **module** phone2(hookState, usr2phone, net2phone);
  priority phone < phone3
  trans
    usr2phone?$dial(X)$, hookState = $dialtone$ $\rightarrow$ hookState := $wait$, phone2net!$call(X)$;
    usr2phone?$dial(X)$, hookState != $dialtone$ $\rightarrow$;
    net2phone?$accept$, hookState = $wait$ $\rightarrow$ hookState = $connected$;
    net2phone?$accept$, hookState != $wait$ $\rightarrow$;
    net2phone?$busy$, hookState = $wait$ $\rightarrow$ hookState = $busy$;
    net2phone?$busy$, hookState != $wait$ $\rightarrow$;
}

As the next step, we incorporate incoming calls into this specification.

type FromNetwork2 = **augment** FromNetwork1 **by** $\{call, hangup\}$;
type ToNetwork1 = **augment** ToNetwork **by** $\{accept, busy, hangup\}$;
type HookStatus4 = **augment** HookStatus3 **at** $offhook$ **by** $\{disconn\}$;
module phone4(hookState, usr2phone, net2phone, phone2net) {
  chan net2phone: FromNetwork2; phone2net: ToNetwork1;
  var hookState: HookStatus4;
  var phone: **module** phone3(hookState, usr2phone, net2phone, phone2net);
  priority phone.phone < phone4; /* phone2 < phone1 */
  trans
    net2phone?$call$, hookState = $idle$ $\rightarrow$ phone2net!$accept$, hookState := $ring$;
    net2phone?$call$, hookState != $idle$ $\rightarrow$ phone2net!$busy$;
    net2phone?$hangup$, hookState = $connected$ $\rightarrow$ hookState := $disconn$;
    net2phone?$hangup$, hookState = $ring$ $\rightarrow$ hookState := $idle$;
    net2phone?$hangup$, hookState != $ring$, hookState != $connected$ $\rightarrow$;
    user2phone?$replaceHandSet$, hookState = $wait$ $\rightarrow$ phone2net!$hangup$, hookState := $idle$;
    user2phone?$replaceHandSet$, hookState = $connected$ $\rightarrow$ phone2net!$hangup$, hookState := $idle$;
}

Note that in this derivation step, the transition rules in this module are not given priority over those in phone3. Thus, all the rules in phone3 are applicable in phone4. In successive refinements, it is easy to add more functionality to this basic model, such as call screening, caller-id, using the type derivation and module derivation operations.

4 Incremental Validation

As mentioned in the introduction, one of the costliest aspects of validation of evolving systems is the need to reestablish all the properties after each modification. We can significantly reduce this cost through *incremental validation*, wherein we spend our effort mainly in establishing the new properties of the modified system, while being able to carry over (most of) the properties established before the modification. In this section, we outline how our language mechanisms can be used to automate incremental validation.

We view validation as a kind of language containment problem, as in the automata-theoretic approach, first proposed in [15]. Specifically, we view the system as an automata A_M, with its *behaviors* (as defined on page 10) corresponding to the set of strings $\mathcal{L}(M)$ (over the alphabet consisting of the set of states of M) accepted by the automaton. In order to verify a property P we construct another automaton A_P that accepts the language $\mathcal{L}(P)$ of all behaviors that *violate* the property. Validation thus amounts to establishing the emptiness of $\mathcal{L}(M) \cap \mathcal{L}(P)$. We remark that currently, RL permits us to express only $*$-automata, whereas most automata-theoretic approaches to verification make use of ω-automata or Büchi automata. However, the discussions in this section on incremental validation are orthogonal to this issue, and hence would be applicable for extensions of RL to support ω-automata as well.

Kurshan [11] proposed a verification method based on *homomorphic reductions* that can be used to deal with step-wise refinement. After modifying an existing specification M to get M', we can go through the following steps to carry over a property P from from M to M'. We specify a mapping h from $\mathcal{L}(M')$ into $\mathcal{L}(M)$. We also identify a property P' for M' such that $h(\mathcal{L}(P')) \subset \mathcal{L}(P)$. ($P'$ can often be mechanically derived P, based on inverting h.) Now, we can see that

$$\mathcal{L}(M) \cap \mathcal{L}(P) = \phi \Rightarrow h(\mathcal{L}(M')) \cap h(\mathcal{L}(P')) = \phi \Rightarrow \mathcal{L}(M') \cap \mathcal{L}(P') = \phi$$

The first implication is because $h(\mathcal{L}(M')) \subset \mathcal{L}(M)$ and $h(\mathcal{L}(P')) \subset \mathcal{L}(P)$.

The second implication follows from the fact that if the images of $\mathcal{L}(M')$ and $\mathcal{L}(P')$ under h do not have any common elements, then the two sets themselves cannot have common elements.

The notion of reduction mapping is similar to the ideas of *protocol projection* [12] and abstraction mappings [7]. However, the difficult aspect of all these approaches is that the mapping h needs to be constructed by the programmer, and is often the most time-consuming and difficult component of the verification approach. In the rest of this section, we discuss how the derivation constructs in RL can be used to automate this process.

4.1 Our Approach

The type and module derivation constructs in RL define a natural mapping h_s from the states of the refined specification M' to the original specification M. Based on this mapping, we can define a mapping h_r from transition rules in M' to those in M such that whenever a transition R' from a state S'_1 to S'_2 can be taken in M', the transition $h_r(R')$ can be taken to go from $h_s(S'_1)$ to $h_s(S'_2)$ in M. It is easy to see the existence of the mappings h_s and h_r imply the existence of the mapping h with the desired properties, provided $h_s(S'_0) = S_0$, where S'_0 and S_0 denote the initial states of M' and M respectively.

The mapping h_s is specified as follows. For each variable v in M' that has a type T' derived from its type T in M, we replace the value of v by $\mathcal{M}_{\mathsf{T}' \to \mathsf{T}}(v)$. More formally,

$$\begin{aligned} h_s(S_{M'})(v) &= \mathcal{M}_{\mathsf{T}' \to \mathsf{T}}(S_{M'}(v)), \text{ if } \mathcal{M}_{\mathsf{T}' \to \mathsf{T}}(S_{M'}(v)) \text{ is a proper element} \\ &= \text{ any proper value derived from same value as } \mathcal{M}_{\mathsf{T}' \to \mathsf{T}}(S_{M'}(v)), \text{ otherwise} \end{aligned}$$

Every rule R that is present in M as well as M' maps onto itself under h_r. For others, we can proceed to check the existence of a h_r satisfying the conditions specified earlier for each state S' of M' — a task that is equivalent to a state-space search of M' and hence defeats the whole purpose of incremental validation. In order to avoid this, when we consider a rule R', we group together all the states that agree on the variables in R'. Typically, the number of such groups is much smaller than the entire state space. Considering each rule R' in M' and each group $\mathbf{S}'_{R'}$ of states that agree on values of variables in R', we identify a rule R in M such that

1. if R' takes $\mathbf{S}'_{R'}$ to some $\mathbf{S}'$ then R takes $h_s(\mathbf{S}'_{R'})$ to $h_s(\mathbf{S}')$.

2. For any R'_1 and R'_2 in M', if they are mapped onto R_1 and R_2 respectively (in M) by the previous step, and R'_1 does *not* have higher priority over R'_2 then R_1 does not have higher priority over R_2.

The second step ensures that whenever a rule R'_2 is enabled, its image in M is enabled.

The above approach yields a procedure for automatically checking for the existence of a mapping h from M' to M. The procedure may have to search through the h_s mappings that correspond to different choices for mapping augmented values. The procedure may also have to consider different mappings between the rules in M' and M. However, we can make the procedures far more efficient by using the mapping between rules to guide the mapping for augmented values and using heuristics to cut down the search.

4.2 Improvements to the Basic Approach

Two classes of improvements can be made to the basic approach so that the property of interest can be carried over even when the basic approach fails to identify the h_r mapping. In the first class, we exploit the property to be carried over so as to eliminate certain rules in M' that do not map onto rules in M. For instance, we can eliminate rules that can never be taken in any history generated by the property automaton. We can also eliminate rules whose presence or absence does not affect the property. As a concrete example, consider a rule R' in M' that introduces transitions between two refined states. Thus $h_r(R')$ will be a rule that corresponds to a transition from a state onto itself, i.e., a self-loop. Typically, such loops are absent in M, which leads to a failure of the basic approach. However, such self-loops do not affect satisfaction in the context of most properties (liveness properties in the context of certain fairness criteria and safety properties). This is especially true in the context of interleaving semantics, where there is little distinction between making a transition to the same state and not making any transition at all. Therefore, we can often eliminate such rules and establish a mapping only for the others.

In the second case, we can relax the condition on the mappings so that we can look at sets of transitions at a time rather than individual transitions. For instance, if every set of transitions in M' is covered by a corresponding set in M, our correctness properties can still be carried through. When such techniques also fail, it appears that it may still be possible to avoid complete state-space search for certain classes of properties, by exploiting the fact we need only consider those paths corresponding to the rules that were not covered by the mapping. Further research is required to formalize this idea.

4.3 Illustration of Incremental Validation

A class of modifications that often takes place in RL is as follows. We refine some state variables, and strengthen existing rules so that the tests or assignments in them refer to the refined values rather than the unrefined values. Moreover, new transition rules may be added to make transitions between the refined states. For instance, the successive specifications phone1 through phone4 all make use of this kind of transformation. The above technique allows us to carry over all the properties after each such transformation. We illustrate this by defining a mapping h_r from phone3 to phone2. In order to compactly describe the mapping, we first present the image of the new rules in phone3 under a mapping h_1 and then show the existence of a mapping h_2 satisfying the above characteristics from the image to the rules in phone2. (Thus, $h_r(R') = h_2(h_1(R'))$ for every rule R' in phone3.) The mapping h_1 is once again derived from the mapping on states. The mapping of values added through refinement is straight-forward, but there are some choices for the mapping of augmented values. We map a state assigning $dial(X)$ to usr2phone in phone3 to a state that assigns ϵ to the same variable in phone2. Similarly, for the variable hookState, we map *wait* to *dialtone* and *busy* to *connected*. Based on this mapping, we can now map the expressions in the rules. We can also delete the variable phone2net newly introduced in phone3 to get

usr2phone $= \epsilon$, hookState $= dialtone \;\; \rightarrow$ usr2phone $:= \epsilon$, hookState $:= dialtone$;
usr2phone $= \epsilon$, hookState $!= dialtone \;\; \rightarrow$ usr2phone $:= \epsilon$;
net2phone?$signal$, hookState $= dialtone \;\; \rightarrow$ hookState $:= connected$;
net2phone?$signal \;\; \rightarrow$;

The first four rules in phone3 map to the above four rules, whereas the fifth and sixth rules map onto the third and fourth rules above. Note that when we map the fourth rule, the condition hookState! $= wait$ disappears. This is because such a condition would be enabled in states that assign one of the values $\{idle, ring, dialtone, connected, busy\}$ to hookState. Mapping this back to phone2, the condition is enabled in states that assign one of $\{idle, ring, dialtone, connected\}$ to hookState, i.e., for every value of hookState.

From the above image under h_1, we now specify the mapping to rules in phone2. The first and second rules take a state to itself, and hence can be ignored, as discussed before. The third and fourth rules above map onto the third and sixth rules respectively in phone2.

5 Compilation of RL

In this section we describe algorithms for compiling RL specifications into FSMs and present an overview of implementation of our prototype compiler and simulator for RL. The compilation enables us to reuse traditional validation algorithms (based on FSMs) for establishing properties. Our compilation algorithms can also be used to do a source code transformation that collapses a module that is described through a sequence of module derivations into a single module description. As mentioned earlier, this transformation is useful to understand specifications that have gone through several refinements.

Input programs are first parsed and other routine aspects such as elimination of channel variables are taken care of. Following this, module instantiation takes place. To instantiate a module M, we first instantiate its component modules and bring their rules together with those of M. Priority relationships between rules are held in a separate table. Without the module instantiation step, we would have to search through all the module instances to identify the transition(s) that can be made. By bringing together all the rules, we can build efficient automata [14] for quickly testing which rules are executable.

Following module instantiation, we translate the resulting rules into our target language. Our target language, designed to describe FSMs, supports only integer data types and has no notion of priority. It supports several kinds of branching constructs: if-then-else, the **switch** construct similar to that of the C language, the **indeterminate** construct that denotes an n-way non-deterministic branch, and the **ordered** construct, that denotes the fact that the children branches are to be tried in order. Given this target language, the important aspects of code generation are (a) mapping of source language data types into target language (integer) data types, and (b) construction of the transition rule automaton. We describe each of these aspects below.

5.1 Mapping Source Language Data Types into Target Language Data Types

In this phase, we first map structured types onto a set of integer variables, and then encode enumerated values by integers. For the mapping, we associate an array of locations to store a structured variable. For each node of the structured data, the mapping specifies the index where the value of the node is stored. Observe that it is acceptable to have the location corresponding to a node depend upon the values of its ancestor nodes, since we will always examine the ancestor nodes before arriving at this node. However, the location cannot be dependent on values of non-ancestor nodes, since the pattern-matching construct implies that we may not always have inspected such nodes. Consider:

$$\begin{array}{l} \textbf{type } \mathsf{T1} = \{a, b, c\} \\ \textbf{type } \mathsf{T2} = \{d(\mathsf{T1}, \mathsf{T1}), e(\mathsf{T1})\} \\ \textbf{type } \mathsf{T3} = \{f(\mathsf{T2}, \mathsf{T1}), g(\mathsf{T1}, \mathsf{T2})\} \end{array}$$

Since T1 is not structured, no mapping is generated. For T2, we map the root and the children values at offsets 0, 1 and 2 respectively. For type T3, the root is stored at offset 0, and the locations of the child nodes will depend on the value of the root node. For instance, in the term $f(e(a), b)$, f is stored at 0, e at 1, a at 2, and b at 4. Note that b is not stored at the location immediately after a, since this location would be occupied by the second child of of d, had d been in the place of e. Similarly, in the term $g(a, d(b, c))$, the values g, a, d, b and c are stored at locations $0, 1, 2, 3$ and 4 respectively.

The encoding of enumerated values becomes complex when we need to support type derivation efficiently. Consider the comparison $\mathsf{v} = offhook$ where v has the type HookStatus3 shown on page 6. The test must succeed iff the value of v is one of $connected, dialtone$ or $wait$. The refinement step thus requires us to make several comparisons in place of one. However, by assigning contiguous codes to all refinements of a value,we can reduce the number of comparisons to two. The table below shows the two comparisons in the above example, where C_{val} denotes the integer code assigned to val. In order to assign contiguous codes, after encoding a value V in a type T, we recursively check through the refinements of T, and assign contiguous codes to them. The next value in T gets encoded by an integer that is greater than the code assigned to any refinement of V.

The table below shows the translation of assignments and comparisons based on the above mappings. In the table, u has the type HookStatus, and

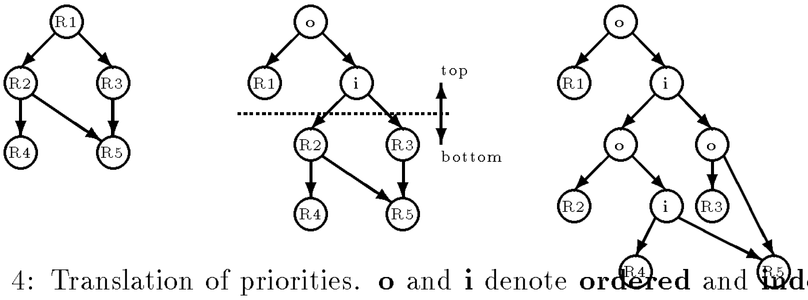

Figure 4: Translation of priorities. **o** and **i** denote **ordered** and **indeterminate** nodes

the notation M_{var} denotes the memory location for storing a variable var, and $*$ denotes dereferencing.

Source Construct	Target Language Translation
$\mathsf{v} = offhook$	$*M_{\mathsf{v}} \geq C_{connected} \wedge *M_{\mathsf{v}} \leq C_{wait}$
$\mathsf{v} = wait(X)$	**if** $(*M_{\mathsf{v}}) = C_{wait}$ **then** $M_X := *(M_{\mathsf{v}} + 1)$
$\mathsf{u} := \mathsf{v}$	**switch** $(*M_v)$ $C_{connected}, C_{dialtone}, C_{wait}$: $M_u := C_{offhook}$ **default:** $M_u := *M_v$

5.2 Construction of Transition Rule Automaton

In this phase, the priority information is integrated into the rules by using the **ordered** and **indeterminate** constructs. To do this, we start from the roots of the (minimal) directed acyclic graph representing the partial order relationship given by the priorities. At any time during this procedure, we would have converted some "top portion" of the dag into **ordered** and **indeterminate** constructs (see Figure 4). We then select a rule R with maximal priority from among the rules below this top portion. The rules immediately below R are combined using an **indeterminate** node, which is then combined with R using an **ordered** node. Figure 4 shows two such steps.

Note that the automata construction does not lose the structure of the rules. Therefore, if we avoid translating comparisons and assignments, it is possible to generate (from the automaton) a set of rules that closely correspond to the rules in the original specification. However, since the module boundaries have been lost, we have a transformed program wherein the net effect of several module derivation steps is captured in a single specification.

5.3 Discussion of Prototype Implementation

The compiler is written in Standard ML of New Jersey. The whole compiler consists of about 6000 lines of SML code, with about 4000 in the preprocessing phase and 2000 in the code generation phase. All of the components described above have been incorporated into the compiler, with the exception of test reordering. The current performance of the prototype compiler is quite good. It is fast and generates compact code. A simulator (about 300 lines of C and SML code) for the resultant FSMs has also been implemented. The simulator first converts the finite state machine into C-code, which is then compiled and run. The compilation of C-code is somewhat slow, but execution speed is already quite fast: about 10^4 transitions per second for some sample specifications on a Sun Sparc LX machine. We are investigating further speed improvements using test-reordering.

6 Concluding Remarks

In this paper, we presented a specification language RL for modelling evolving systems. RL provides two principal mechanisms, namely, type and module derivation, that operate in conjunction to enable modelling of incremental changes to a specification. The design of these mechanisms must minimize the potential conflict between the expressive power of the mechanism and the simplicity of semantics. We illustrated the power and convenience of RLthrough a set of examples and then developed a methodology for incremental validation. The important aspect of our methodology is that it uses RL mechanisms for refinement and evolution to *automatically* carry over properties, without requiring human intervention. Finally, we presented algorithms and results of a preliminary implementation for compiling RL programs into FSAs. Since such FSAs can be efficiently analyzed for purposes of "validating from scratch," our language support for refinement and evolution is achieved *without* compromising efficiency or requiring rediscovery of previously known algorithms for validation. We are currently investigating means to further enhance the power and convenience of our mechanisms for supporting evolution.

References

[1] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, *Information and*

Computation, 98(2), 1992.

[2] F. Belina and D. Hogrefe, The CCITT Specification and Description Language SDL, *Computer Networks and ISDN Systems 16(4), 1988.*

[3] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language Lotos, *Computer Networks and ISDN Systems 14, 1987.*

[4] S. Budkowski and P. Dembinski, An Introduction to Estelle: A Specification Language for Distributed Systems, *Computer Networks and ISDN Systems 14, 1987.*

[5] E.M. Clarke, E.A. Emerson and A.P. Sistla, Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications, *ACM TOPLAS, 8(2), 1986.*

[6] D. Comer and R. Sethi, Complexity of Trie Index Construction, *FOCS*, 1976.

[7] E.M. Clarke, O. Grumberg and D.E. Long, Model Checking and Abstraction, *POPL '92.*

[8] D. Harel, Statecharts: A Visual Formalism for Complex Systems, *Sci. of Comp. Prog., 1987.*

[9] C.A.R. Hoare, Communicating Sequential Processes, *CACM 21(8), 1978.*

[10] G.J. Holzman, Design and Validation of Computer Protocols, *Prentice Hall, 1991.*

[11] R.P. Kurshan, Automata Theoretic Verification of Coordinating Processing, *Berkeley Lecture Notes (unpublished), 1991.* Also, Verification of Concurrent Processes: The Automata-Theoretic Approach, *Princeton University Press (to appear), 1994.*

[12] S.S. Lam and A.U. Shankar, Protocol Verification via Projections, *IEEE Transactions on Software Engineering, 10(4), July 1984.*

[13] J.L. Peterson, Petri Net Theory and the Modelling of Systems, *Prentice Hall, 1981.*

[14] R. Sekar, R. Ramesh and I.V. Ramakrishnan, Adaptive Pattern Matching, *ICALP '92.*

[15] M.Y. Vardi and P. Wolper, An Automata-theoretic Approach to Automatic Program Verification, *Proc. on Logics in Computer Science, 1986.*