A Framework for Multi-Notation Requirements Specification and Analysis

Nancy A. Day
Department of Computer Science
Oregon Graduate Institute
20000 N.W. Walker Rd.
Beaverton, OR, 97006 USA
nday@cse.ogi.edu

Jeffrey J. Joyce
Intrepid Critical Software Inc.
P.O. Box 39207
Vancouver, BC, V6R 4P1 Canada
joyce@intrepid-cs.com

Abstract

Many organizations desire the convenience of using multiple notations within a requirements specification. Rather than using separate tools for each notation, we advocate combining the parts semantically for tool-based analysis. We describe a framework for integrating notations from four distinct categories, namely, "models", "events", "actions", and "expressions". The categories allow us to view the notations independently, but in a manner whereby they can be combined to create a specification. The categories are implemented as types in higher-order logic. Typechecking ensures conformance to the rules for combining notations. Our choice of higher-order logic as a base formalism allows the framework to support notations with uninterpreted constants. With our framework, it is possible to use new combinations of notations without changing existing notations or rebuilding formal analysis tools such as model checkers.

1. Introduction

Many organizations and individuals use different notations for expressing requirements of different aspects of a system. DeMarco's structured analysis methodology is an early example of an integrated collection of informal notations [20]. Some languages, such as the Requirements State Machine Language (RSML) [32], the Software Cost Reduction (SCR) language [27], and the Unified Modeling Language (UML) [40], can be viewed as collections of notations. The ability to use multiple notations allows a specifier to choose the notation most appropriate for describing each aspect of a system's behaviour. For example, tables are often a more natural choice for describing decision logic than a state diagram. In this paper, we describe a framework for creating formal analysis tools for multi-notation specifications.

Notational approaches usually begin with a new concept for the representation of a particular type of behaviour or structure. However, the approach requires additional elements to create a language for describing all aspects of a system. For example, the variants of statecharts [22] are all based on the central concept of representing complex state behaviour in terms of hierarchical, concurrent state-transition diagrams. Some differ on peripheral matters, such as the notations used to specify transition labels. Rather than being able to "plug in" a different notation for transition labels, a new variant was created. In our framework, we view these component notations independently and provide the ability to "plug in" notations to create new combinations of notations, and the analysis tools for multi-notation specifications.

Because some analysis properties rely on information described in different notations in a specification, there is a need for analysis to span multiple notations. Supporting analysis tools have generally been developed for the complete approach, rather than maintaining the independence of the central concept from the additional elements (e.g., [25, 31]). These tools do not typically provide the user with the option to substitute one set of additional elements for a different set. Instead, tool support has to be created for each new composite of multiple notations. While an organization will likely wish to adopt a fixed collection of notations for a particular project, different combinations will be appropriate for various projects.

Building formal analysis tools, even for a single notation, is a time-consuming task. Translation into the input notation of an existing analysis tool is a common approach either for a single notation ([6, 5, 8, 4, 25]) or for multiple notations ([43, 7, 38]). Translation bridges the gap between notations developed for their readability and comprehensibility, and notations developed because they can be analyzed. A translator must be created for each new notation, and for each additional analysis tool. The translator is software that has to be checked for conformance with the semantics of the notation. Also, there is often a mismatch between the expressiveness of the requirements notation and the analysis tool's input notation. For example, many automated tools accept only specifications with finite state spaces. This mismatch requires that the translation includes an abstraction step.

Our framework consists of a set of standard interfaces that capture the semantic information needed about individual notations to analyze multi-notation specifications. As a starting point, we concentrate on model-oriented, functional requirements notations where multiple parts of the specification do not describe overlapping behaviour. A model-oriented notation is one that contributes to the description of a system in terms of a relationship between a current state and its immediate successor(s). There are automated formal analysis methods, such as model checking [14], that apply to specifications in these notations.

As statecharts have formed the basis for many model-oriented requirements methodologies, we begin by teasing apart the components of the statechart notation. We find *expressions*, *events*, and *actions*, which are all used to label transitions. We also need a name for the overall governing state-transition part; we call this a *model*. Statecharts are an example of a notation that belongs in the model category. We create standard, independent interfaces for each of these "categories" (expressions, events, actions and models). The interface describes the semantic information that a notation should provide. A notation that provides the information associated with a particular category is considered a notation of that category. The semantics of a particular notation may rely on the interface provided by a notation in another category. For example, the meaning of a statechart depends on the meaning of the event notation used. We call these slots, where one notation can be combined with another notation, *join points*[1]. With these interfaces, we make notations independent of each other, but still able to depend on the fixed interface supplied by another notation. The insights in this paper lie in the choice of elements for the interfaces.

As an implementation environment for our framework, we use higher-order logic (hol). The categories are described as types in hol. Notations are embedded in hol in a form that meets the requirements of a category's interface. The meaning of the multi-notation specification is its meaning in hol, which can be subjected to analysis. We work with hol outside of a theorem proving environment, and apply automated analysis to our multi-notation specifications.

In hol, we find a number of desirable features. First, there is a type system that enforces the interfaces. If the interfaces are used correctly (i.e., the specification is well typed) then the multi-notation specification is well defined and can be analyzed. Second, hol is a general-purpose formalism for writing the semantics of notations. The categories restrict only the type of the semantics, not their style. Third, we can evaluate the semantics directly using symbolic functional evaluation to produce input for automated analysis without needing to write a translator [18]. Fourth, hol is an expressive "intermediate" formalism for linking with a range of automated analysis tools. The representation of the notation in hol is equivalent to its original form – no abstractions are applied in the embedding process. Because hol is expressive, it places few limits on the types of analysis that can be performed. For example, we can apply model checking of a finite state system, but also theorem proving, and other future analysis techniques. Finally, hol is a logic that can itself be used as a notation, allowing the use of uninterpreted constants in a specification.

By looking at these categories generally, we find interesting combinations that might not have been tried. For example, logical expressions, including uninterpreted constants, can be used as expressions in a statechart. An uninterpreted constant is a symbol that has a type, but no definition. Uninterpreted constants are useful for maintaining a high level of abstraction in a specification and filtering non-essential details. The analysis performed is valid for any definition of the uninterpreted constant. While uninterpreted constants are a source of incompleteness in a final specification, they are useful in incrementally building specifications. With an analysis method that supports their use, specifications can be analyzed before they are complete.

After introducing our notation and some terminology, we present a motivating example for the use of multiple notations, including uninterpreted constants, in a specification of a heating system (Section 4). Section 5 describes the categories: models, events, actions, and expressions. This section describes the use of the categories and treats the types as opaque in order to show how notations can be combined in new and interesting ways. In Section 6, we move under the hood and describe the semantic information a notation must provide to be an element of a category. It is beyond the scope of the paper to show how the semantics in this form can be used directly as input to analysis, but references to this work, and a brief discussion of how the framework has been used in real-world examples, are found in Section 7. These examples are separation minima for aircraft, and an aeronautical telecommunications network. Alternative approaches are discussed in Section 8, and Section 9 concludes the paper.

[1]We borrow this term from subject-oriented programming, where it is used to describe constructs in source code that could be used as the basis for integrating subjects in subject-oriented programming [23]. The term "join point" was first used by W. Harrison and H. Ossher at an aspect-oriented programming workshop in October 1996 [34].

2. Description of our notation

We use the notation S [30], a variant of the higher-order logic used in the HOL theorem prover [21], as the basis for our framework. S includes facilities for the declaration and definition of types and constants similar to Z [41]. It serves as both a specification notation, and as the notation in which to write our semantics. Readers not familiar with higher-order logic can consider it similar to a typed, functional programming language with quantifiers, and with uninterpreted constants. In this section, we briefly introduce the features of S that are used in later sections.

An S specification is a sequence of statements, declaring or defining types and constants. In our framework, the categories are described as types in S that initialize our tool. When these types are used in multi-notation specifications, the S typechecker will flag non-conformance to rules for using notations in combination.

Type expressions of the form, `int -> int`, describe functions. Type expressions of the form, `int # int`, describe tuples. Type expressions of the form, `(int)list`, indicate the parameterized type `list` is instantiated with the type `int`., Type expressions are used in type declarations (prefixed by a "`:`"), constant declarations and definitions. They are also used in type abbreviations, e.g., `:time == num;`. The names of the categories of notations are all type abbreviations for more complex types described in the sections on the semantics.

Constant definitions are given using the `:=` operator, as in `x:=1;`. A constant declaration is the name of a constant followed by a type expression as in `y:num;`.

In constant and type declarations and definitions, we use type variables to indicate the sources of polymorphism in the type. Type variables, if any, are given in a parenthesized list that prefixes the declaration or definition. In constant declarations, and definitions, type variables prefix the constant name, as in `(:ty) id (x:ty) := x;`, for the identity function. In type declarations, definitions, and abbreviations, the type variables follow the "`:`", but precede the type name. In the type abbreviation `:(ty)exp==config->ty;`, the type variable `ty` can be instantiated by any type; in particular, `(bool)exp` is equivalent to `config -> bool`.

We have written our descriptions in `verbatim` font because, with only minor exceptions for clarity, these are exactly the input to our tool.

3. Configurations

A *configuration* is a representation of the values of the variables of the system at a given time. More typically, this is called a state, but because of the confusion with states of a state-transition system notation, we use different terminology. A configuration can be thought of as a "snapshot" of the dynamic behaviour of the system [29]. In a model-oriented notation, all expressions are considered relative to a configuration.

In an operational semantics for models, we reference only two particular configurations: the current and next configurations. The type `config` represents configurations, and we use the parameter names `cf` for the current configuration, and `cf'` for the next configuration.

Variables are mappings from configurations to values. For the most part, the semantic of notations are able to hide this detail from the specifier by applying the configuration argument in the semantics, rather than in the specification. Higher-order functions, i.e., functions that take other functions as arguments, allow us to pass variable names around even though they are actually functions.

In Section 6.2, we will see that expressions, including variables, are functions of configurations. Therefore, expressions of constant value, such as `5`, must be "lifted" to make them functions of configurations. The function `C` is the lift operation for constants and is defined as the constant function:

```
C x cf := x;
```

A preprocessor could insert the needed applications of `C` in front of constant values, but in our examples, we make its use explicit.

4. Example

In this section, we present a motivating example of a heating system specification loosely based on an example found in Booch [9]. We use statecharts, notations for events and actions, as well as decision tables, and higher-order logic to specify the behaviour of the system. We use the term "statecharts" to mean hierarchical, concurrent state-transition diagrams. In our framework, these statecharts can be combined with any event and action notations.

The heating system consists of a controller, a furnace, and multiple rooms, which operate concurrently. The statechart describing the behaviour of a room is found in Figure 1. Every element of the room is parameterized by a room identifier, `i`, so the system can be instantiated with any number of rooms. The parameter `i` is the room identifier. This parameterization is possible without extending the semantics of statecharts because we use S as a base formalism [3]. There are two states that a room can be in: one where it has requested heat (`HEAT_REQUESTED`), and one where it has not requested heat (`NO_HEAT_REQUESTED`). Before it requests heat, it attempts to adjust the valve position until it is fully open. The syntax, event / action, labels each transition with an event and an action. The event event

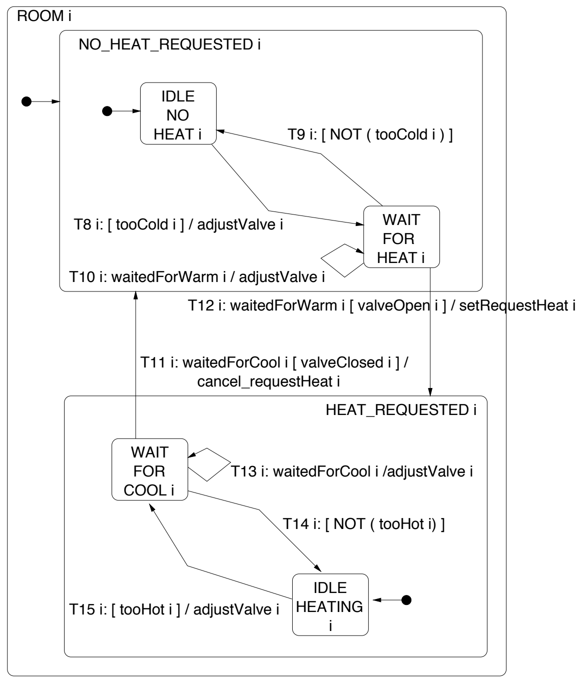
ROOM i
NO_HEAT_REQUESTED i
IDLE NO HEAT i
T9 i: [NOT (tooCold i)]
T8 i: [tooCold i] / adjustValve i
WAIT FOR HEAT i
T10 i: waitedForWarm i / adjustValve i
T12 i: waitedForWarm i [valveOpen i] / setRequestHeat i
T11 i: waitedForCool i [valveClosed i] / cancel_requestHeat i
HEAT_REQUESTED i
WAIT FOR COOL i
T13 i: waitedForCool i /adjustValve i
T14 i: [NOT (tooHot i)]
IDLE HEATING i
T15 i: [tooHot i] / adjustValve i

Figure 1. Specification of a room's behaviour

[...] is an event predicated on a condition. Sometimes the event, condition, or action are omitted.

Each room has a valve that can be in one of three positions: closed, half open, and open. Based on the desired temperature, each room controls its valve position, waits to see if that action has the desired effect, and if not, communicates a request for heat to the controller. The event `waitedForWarm` is defined as a timeout event based on entering the state, and a delay of `warmUpTime`:

```
waitedForWarm i := Tm (En (WAIT_FOR_HEAT i)) warmUpTime;
```

On transition `T8` the action `adjustValve` sets the valve position and is defined as:

```
adjustValve i := Asn (valvePos i) (nextVp i);
```

The value assigned to the valve position depends on the difference between the desired temperature and the actual temperature (from sensor input), as well as the current position of the valve. This type of behaviour is effectively specified using a decision table. Since we are able to use multiple notations within the framework, we can interchangeably use different notations as the expressions in an action. The function `nextVp` is specified by the decision table in Table 1. For other transitions, the actions include simple expressions as in `requestHeat i := true`.

Table 1 is written in a notation that describes expressions. Therefore, decision tables of this form are a notation in the expression category. Given the values of the inputs (desired temperature `dT i`, actual temperature `aT i`, and the valve position `valvePos i`) at the current time, the table defines a value for the next valve position (`nextVp i`). Similar to AND/OR tables [32], the entries in a column must all be true for the next valve position to have the result found at the bottom of that column. A "." in a column means "don't care". Similar to Parnas' generalized decision tables [36], the entries in the rows can be more complex than true and false. The row label is substituted for the underscore in the row entry to form the Boolean-valued expression.

For the purposes of this specification, it is not necessary to specify how the actual temperature is calculated, or how a sensor determines whether the room is occupied or not. We use an uninterpreted function in higher-order logic to specify this function as an expression:

```
aT : room -> (num)exp;
```

The controller (not shown) ensures that the furnace is on if any rooms are requesting heat, and off if no room needs heat. Because we wish to write a specification that can accommodate an unknown number of rooms, the controller statechart includes transitions that trigger on expressions in higher-order logic that include quantifiers. For example, the trigger, `[roomNeedsHeat]`, is defined as:

```
roomNeedsHeat := exists (i:Room). requestHeat i;
```

This example of a heating system specification has motivated the use of statecharts, decision tables, and higher-order logic in combination within one specification. Complete details on the example can be found in Day [16].

5. Categories of notations and their use

In this section, we present the categories model, action, event, and expression. We also describe the way individual notations indicate how they can be composed with other notations by referencing a particular category in a join point. Our goal in this section is to describe the *use* of the framework for creating a multi-notation specification, and how automated typechecking regulates the combinations. The categories are implemented as the following types in higher-order logic: `model`, `action`, `event`, and `exp`. If the specification passes typechecking, it will have a well defined meaning for analysis.

We use textual representations of the notations. A textual representation can match the structure of a graphical or tabular representation. The textual representation can often be created automatically by accessing the internal representation of a specification in a GUI-based toolkit.

Each *keyword* of a notation is a function in higher-order logic. In theorem proving terminology, this is known as a shallow embedding of the notation [10]. Join points to other

dT i - aT i	_ < C -5	C -5 <= _ AND _ < C -2	C -5 <= _ AND _ < C -2	C -2 <= _ AND _ <= C 2	C 2 < _ AND _ <= C 5	C 2 < _ AND _ <= C 5	C 5 < _
valvePos i	.	_ = C OPEN	_ = C HALF	.	_ = C CLOSED	_ = C HALF	.
nextVp i	C CLOSED	C HALF	C CLOSED	valvePos i	C HALF	C OPEN	C OPEN

Table 1. Valve position

notations are found in the arguments to keywords. The parser and typechecker of higher-order logic serve as a universal parser and typechecker for all notations. The framework is extensible, in that no change is needed to a keyword to use it with a different notation of the category at a join point.

5.1. Models

A model denotes a relation between two configurations. The relation indicates that moving from the first configuration to the second is a legal step in the system. Examples of notations in the model category are statecharts, mode-transition tables in SCR, and Petri nets [37].

In the framework, our variant of statecharts is represented using the keyword `Sc`, which has the following type:

```
Sc : sc_structure -> model;
```

A textual representation capturing the hierarchical arrangement of the states and transitions of a statechart are elements of the type `sc_structure`. Part of a statechart is a list of transitions, of type `trans`:

```
: trans ==
    transName # stateName # event # action # stateName;
```

A transition is a five-tuple consisting of a label for the transition (e.g., `T10`), a source state name, an event, an action, and a destination state name. The event and action elements of a transition are join points to any event and action notations, respectively. Typechecking enforces conformance to the framework. For example, if a specifier attempts to use an event notation as the fourth element in the representation of a transition, typechecking would flag this non-conformance as an error.

5.2. Actions

An action denotes a change in the values of variables. The simplest example of an action is an assignment statement. Generating primitive events is another commonly used action. But with the flexibility of the framework, more complex action notations are also possible – perhaps a subset of a programming language.

For example, the `Asn` action, used in the definition of `adjustValve` for transition `T8` of the room statechart, has the following type signature:

```
(:ty) Asn : (ty)exp -> (ty)exp -> action;
```

From this type signature of `Asn`, we know `valvePos i` and `nextVp i`, used in the definition of `adjustValve`, must be written in expression notations, and because they use the same type variable, the return types of the two expressions must match. Table 1 specifying `nextVp i` is written in an expression notation.

SCR event tables are another example of an action notation. They describe what function is to be performed in terms of the current mode and the events.

Actions describe the relationship between the value of a particular variable in two configurations. They have a different required interface than models. Notations in the action category provide information to resolve race conditions, or to enforce a constraint that if a variable is not explicitly changed, it keeps its current value.

5.3. Events

An event denotes an instantaneous occurrence, such as a change in the value of an expression between two configurations. Event notations, such as that of statecharts, usually include primitive events, changes in conditions, events in combination with conditions, and timeouts, as well as multiple events in combinations.

For example, the type signature for the `Tm` keyword is:

```
Tm : event -> (num)exp -> event;
```

`Tm` is a keyword of a notation in the event category. `Tm` takes an element expressed in any notation in the event category, following by a numeric expression, and returns an event.

Other examples of events are the `@T(expression)` and `@F(expression)` events of SCR.

5.4. Expressions

An expression denotes a value in a configuration. Expressions can be written in a variety of notations, such as higher-order logic, AND/OR tables, and decision tables. Anything that takes a configuration and returns a value is an element of the expression category.

The decision table of Table 1 is represented textually, using keywords such as `Table` and `Row`. The keyword `Table` has the type signature:

```
(:ty) Table :(row)list -> ((ty)exp) list -> (ty)exp;
```

This type signature shows that the elements of the result row (the second argument to `Table`) can be written in any notation that produces expressions. The rows are composed of expressions, i.e., join points to other expression notations, such as other tables or logical expressions.

6. Semantics

For analysis, we determine the meaning of a multi-notation specification from the meaning of its independent parts. For example, the meaning of an event is used in determining the meaning of a model. Consequently, the insight comes in finding the right collection of information that allows notations in the categories to be independent, but still work together. In this section, we describe the elements of each category's meaning prescribed by the interfaces, along with a few examples of the semantics of notations in S.

In our higher-order logic implementation, the types of the categories are elaborated to specify the information that the notation must provide to satisfy the interface. For example, the type `event` becomes a type abbreviation for a tuple type with the five components necessary to describe the meaning of an event. A notation such as statecharts, then uses the meaning of its events and actions to determine its own meaning as a model.

6.1. Models

A model is a relation between two configurations. Thus, the type of its meaning is a mapping from a pair of configurations to a Boolean value:

```
: model == (config # config) -> bool;
```

A description of the notation's meaning in this form is an operational semantics, and is suitable for input to model checking analysis. The semantics of the notation that result in this relation may be quite complex. In this paper, we can only provide the flavour of the semantics of the statecharts variant that we have implemented [15, 16]. These semantics are independent of the particular event and action notations used.

In a statechart, a set of transitions are followed in a step. Following the transitions results in the system moving from one set of states to another, and updating the values of the variables. It is also necessary to update the status of events because of the passage of time and the actions that have occurred in this step.

To tackle the complexity, we separate the semantics into three parts, which update the set of states, events, and actions separately. Updating the status of events and actions, relies on the semantics of the event and action notations.

To illustrate how the semantics of one notation rely on the interface of a notation of another category, we consider the semantics for whether a transition is enabled or not. This predicate is true if the system is currently in the source state of the transition, and if the event occurred. All notations in the event category, which will be described in Section 6.4, must provide the predicate `Occurred` for its events. This function takes arguments that are the event, a label to identify the event (in this case based on the transition name), and a configuration:

```
Enabled (tr:trans) cf :=
  InState (transSrc tr) cf and
  Occurred (transEvent tr) (label (transName tr)) cf;
```

6.2. Expressions

An expression takes a configuration and returns a value:

```
: (ty)exp == config -> ty;
```

Any logical statement of this type is an expression, including ones with uninterpreted constants, and quantifiers[2]. Thus, we can use higher-order logic expressions in combination with statechart specifications, as in the heating system example presented in Section 4.

Another notation for expressions is decision tables. The textual representation of the notation used in Table 1 uses the keyword `Table`. The keyword `Table` is defined with its semantics to return an expression [19]. Each of the cells of the table are also expressions, meaning any expression notation can be used in those cells. Tables can reference other tables. For example, the result of a table can be specified by another table.

6.3. Actions

The meaning of an action needs to describe not only the changes in variables, but also what variables are being modified, and what happens if the action is not taken. This information allows model notations to resolve race conditions where multiple actions affect the same variable. It also allows the models to constrain variables to keep the value they have in the current configuration if they are not explicitly changed by an action. The meaning of an action is a three-tuple. The type `action` is a type abbreviation for,

```
: action == string #
            ((config # config) -> bool) #
            ((config # config) -> bool)   ;
```

where the variable being modified is a string, the change that occurs is a relation between two configurations, and the constraint on not changing the value of the variable is also a relation.

For example, using lambda notation in S, an assignment statement is defined as:

[2]The meaning of quantification is extended to be considered relative to the configuration.

```
(:ty) Asn (var:(ty)exp) (expr:(ty)exp)
    := ( name var,
         \(cf, cf'). var cf' = expr cf,
         \(cf, cf'). var cf' = var cf   );
```

The Asn keyword takes two expressions as arguments, and returns an action. The first part of the three-tuple is the name of the value being modified. The second part describes its behaviour if the assignment statement is taken. In this case, we constrain the value of the first argument in the next configuration (cf') to be the value of the second argument in the current configuration (cf). Using the interface descriptions, we know that all expressions take a configuration as an argument and return a value. The third part of the result of Asn describes what happens if no action modifies the value: it remains the same.

6.4. Events

Events have the most complicated interface of the four categories. Giving meaning to an event may require knowing if a nested event is occurring in this step. Thus, events must be independent of each other to mix and match event notations. We illustrate the semantics of the timeout event.

A timeout event has the form Tm ev n, with the meaning that the event ev occurred n time units in the past. We call the n parameter, the delay. Our semantics use an auxiliary counter to record how long it has been since the event last occurred. This counter is an uninterpreted function TmCounter, which can be viewed as an array:

```
TmCounter : index -> (num)exp ;
```

where the first argument can be viewed as an index to the array. Therefore, an element of the array is the time an event last occurred. The semantics constrain the value of this function in each step. The semantics of Tm do not assume any finite bound on the timer.

The indices to this array must be uniquely associated with a particular event. Rather than requiring the specifier to choose these identifiers, we use an identifier scheme based on the nesting of events. In the semantics, the function Sub modifies the index based on the hierarchy. Using this scheme, the same timeout event used on different transitions will have multiple identical entries in the counter array.

The meaning of an event contains five components:

occurred: takes a configuration and indicates if the event occurred in that configuration (i.e., the event occurred in the previous step). A timeout event occurred if the value of the counter is equal to the amount of time the timeout is supposed to wait, i.e., the value of the delay in the current configuration.

```
TmOccurred index n cf :=
    TmCounter index cf = n cf ;
```

Because the value of the delay is evaluated relative to a configuration, if it is a variable, its value may change. This part of an event's meaning is used by a model notation to trigger a transition.

occurs: takes two configurations and indicates if the event is occurring in this step. For a timeout, this means the delay matches the value of the counter function in the next configuration cf':

```
TmOccurs index n (cf,cf') :=
   TmCounter index cf' = n cf';
```

update: takes two configurations and constrains the value of the counter functions in the next configuration to capture the history of an event. There are two cases: 1) the event of the timeout occurs in this step, and the counter is reset; 2) the event does not occur, and the counter is incremented. To determine if the event of the timeout occurs, we call the function Occurs defined for the event of the timeout. We must also update the counter for the event of the timeout, which could potentially consist of timeouts; we do this using the update function for the event of the timeout.

```
TmUpdate ev index (cf,cf') :=
    ((Occurs ev (Sub index) (cf,cf') and
      TmCounter index cf' = 0)    or
     (not(Occurs ev (Sub index) (cf,cf')) and
       (TmCounter index cf' =
          (TmCounter index cf + 1 )))) and
    Update ev (Sub index) (cf,cf');
```

This part of the meaning is used by the model to update the status of events.

occurred at init: takes a configuration and indicates if the event occurred in the initial configuration. We choose the semantics that a timeout occurred in the initial configuration if the value of the delay is 0, and the event of the timeout occurred at initialization:

```
TmOccurredAtInit ev index n cf :=
 (n cf = 0) and (OccurredAtInit ev (Sub index) cf);
```

The occurrence of a timeout at initialization should be rare, since a timeout with delay of 0 should have been specified as a simple event.

init: takes a configuration and initializes the uninterpreted counter function used to maintain history. In the case of timeouts, the value of the uninterpreted function is 0 if the event of the timeout occurred at initialization, otherwise the counter may have any value because we have no knowledge of when the event last occurred. Additionally, the semantics enforce the initialization of the event of the timeout:

```
  TmInit ev index cf :=
   (OccurredAtInit ev (Sub index) cf =
      (TmCounter index cf = 0)
   and Init ev (Sub index) cf;
```

We specify that an event must have these elements by making the type `event` a type abbreviation for a function that maps an index onto a five-tuple:

```
: event ==
   index ->
   ( (config -> bool) #              /* Occurred */
     ((config # config) -> bool) #   /* Occurs */
     ((config # config) -> bool) #   /* Update */
     (config -> bool) #              /* OccurredAtInit */
     (config -> bool) )              /* Init */
   );
```

The name of the accessor functions that isolate each of the parts of the meaning of events are listed in the comments above. The five components of `Tm` are packaged together to define the meaning of `Tm`.

The choice of components of an event's meaning is based on our experience making the different statechart events independent of each other, and independent of the model part of statecharts. Further work is needed to determine whether this list is sufficient for other event notations. Discussions of the semantics of events in other notations can be found in Heitmeyer et al. [26] and Atlee and Buckley [5].

7. Automated formal analysis

To demonstrate how the framework is used for automated formal analysis, we have implemented completeness and consistency checking of tables [24, 26], symbolic CTL model checking [13], and simulation. Manipulating expressions in higher-order logic usually requires the support of a theorem prover. In Day and Joyce [18], we describe how the semantics of the notations can be evaluated outside of a theorem proving environment, using a technique called symbolic functional evaluation. Once evaluated, conservative abstraction methods are used to create a finite model for analysis. Our implementation currently uses abstraction to propositional logic (as in Rajan [39]) and binary decision diagrams [12]. The result of the analysis may be conservative, in that it allows the system to have more behaviours than the original specification. However, by creating a bridge between the world of specification and analysis, it is possible to do some automated analysis even if all types of queries cannot be processed automatically. More about our link between higher-order logic and automated abstraction, including information on how we return results in a readable manner and at the correct level of abstraction, are described in Day, Donat, and Joyce [17].

We applied all implemented types of analysis on the heating system example presented in Section 4. For example, we proved that when enabled, transition `T8` will always be followed in a room (say the kitchen), i.e., at this point there is no non-determinism in the system:

```
prop :=
 ((InBasicState (IDLE_NO_HEAT KITCHEN) cf) and
  (valvePos KITCHEN cf = CLOSED) and
  ((2 < dT KITCHEN cf - aT KITCHEN cf) and
   (dT KITCHEN cf - aT KITCHEN cf <= 5)))
 ImmediateResponse
 ((valvePos KITCHEN cf = HALF) and
  (InBasicState (WAIT_FOR_HEAT KITCHEN) cf));
```

`ImmediateResponse` is an infix operator defined in CTL as: a ImmediateResponse $b = \text{AG}(a \Rightarrow (AXb))$ To prove this property, the model checker uses information found in the multiple notations of the heating system specification. The change in the states is described in a statechart, but the change in the valve position is described in Table 1. Also, the entries in the table are in S. The tool is able to identify automatically that the ranges used in the table are mutually exclusive; a property necessary to ensure the resulting valve position is always `HALF` (otherwise, `nextVp` could return the result of an earlier column). To prove this property we needed to add an environmental constraint that connected the definition of the trigger of transition `T8` (`tooCold`) with the ranges used in Table 1. The property was verified in less than a second on a Pentium II (450MHz, 512 Mb RAM).

We have used our framework for automated analysis of two real-world specifications. One is a description of the International Civil Aviation Organization's separation minima for aircraft in the North Atlantic Region. This specification was formalized using a combination of decision tables and higher-order logic. Uninterpreted functions were used liberally in this specification, in order to maintain a high level of abstraction. The specification was analyzed for completeness, consistency, and symmetry. The analysis showed three places not previously identified where the specification is inconsistent [19].

In a second example, we applied model checking to an aeronautical telecommunications network (ATN) written in our statecharts variant and higher-order logic [3, 16]. The specification consisted of approximately 3100 lines with parameterized components.

8. Related work

Zave and Jackson translate notations into one-sorted first-order logic [43]. They deal with each notation individually, rather than offering guidelines for how notations fit together. Consequently, for each new combination, there is the question of whether the requirements specification has a well defined meaning before analysis can proceed.

Paige describes populating the space of translations between notations, some of which are not one-to-one, as a means of applying refinement rules for one notation to another in multi-notation specifications [35].

We have concentrated on the integration of component notations of state-transition languages, rather than on multiple model notations. For the meaning of multiple models used in a specification, we currently use conjunction [1, 43]. Complementary to our approach is Pezzè and Young's [38] work on the integration of models. They translate models into a common base formalism of hypergraphs, which is similar to a labelled Petri Net. Rules for matching arcs describe how specifications in different notations interact. Our approaches could work together by embedding the hypergraph formalism in higher-order logic, with the appropriate join points to our categories of events, actions, and expressions.

Avrunin et al. [7] use the common formalism of constant slope linear hybrid automata, which are supported by the HyTech verifier [2]. Hybrid automata are constructed from Ada programs, expressions in Graphical Interval Logic (GIL), and regular expressions. Hybrid automata are a form of model, so this integration is also at the model level, and the meaning of the composition is based on the intersection of the hybrid automata.

Bharadwaj and Heitmeyer [8] describe a translation from SCR components into an intermediate form of conditional assignments. They translate this intermediate form to the input languages of SPIN [28] and SMV [33]. In SCR, there is only one layer of state hierarchy, and the specification is assumed to be deterministic before model checking analysis. Other model notations, such as statecharts, have greater complexity in their semantics because of the state hierarchy. In our approach, we formalize the semantics and use them directly in analysis. Furthermore, by using S as our base formalism, the form of the semantics is less restricted than the intermediate formalism used for SCR. We require only that the semantics have the type of the model category.

9. Conclusions

This paper describes a framework of standard interfaces for the use of multiple, model-oriented notations in requirements specification. The framework consists of the four categories: models, events, actions, and expressions. Our choice of these categories works quite naturally in our examples. The integration between the notations is unobtrusive — once the notation has been set up in the framework, the rules on where it can be used in a multi-notation specification are straightforwardly based on types. Individual notations specify links to a category, rather than with another notation. Therefore, new notations can be added to a category and used with the existing set of notations without changing the existing notations. The framework ensures that a multi-notation specification has a well defined meaning. By providing the link between the meaning in higher-order logic and automated analysis, new combinations of notations can analyzed without having to implement a new analysis tool. We have connected our framework with automated techniques, such as completeness and consistency checking of tables, and CTL model checking.

The framework provides a way of understanding how notations may be combined in a rigorous manner. Existing descriptions of the semantics of many notations could be "codified" in higher-order logic to extend the framework to include these notations. The semantics for our variant of statecharts could form the basis for including many state-transition notations in our framework (such as SCR mode-transition tables, and RSML). The work of Wang et al. [42] on formalizing and integrating the dynamic and object models of OMT could aid in bringing OMT into our framework. A means of capturing object hierarchies is necessary for object-oriented notations. Based on the work of Bowen and Gordon [11], which describes a shallow embedding of Z in the object language of the HOL theorem prover, our framework could be extended to include Z [41].

The framework and its supporting tools could be used by organizations wishing to choose a custom set of notations for a particular application, such as air traffic control. An expert would be required to write the semantics of the notations such that they fit into the framework. Individual specifiers would use typechecking to guide them in creating well defined specifications. The resulting specification in the custom choice of notations can then immediately be subjected to automated analysis. Analysis tools need not be modified to accommodate new notations.

10. Acknowledgments

We thank Tony Earnshaw and the reviewers for their comments. The first author is supported by Intel, NSF (EIA-98005542), USAF Air Materiel Command (F19628-96-C-0161), and the Natural Science and Engineering Research Council of Canada (NSERC). This paper is based on results produced by the formalWARE research project supported by the BC Advanced Systems Institute, Raytheon Systems Canada, and MacDonald Dettwiler.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. Technical Report 118, Digital Systems Research Center, December 1993.

[2] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. on Soft. Eng.*, 22(3):181–201, March 1996.

[3] J. H. Andrews, N. A. Day, and J. J. Joyce. Using a formal description technique to model aspects of a global air traffic telecommunications network. In *FORTE/PSTV*, 1997.

[4] M. M. Archer, C. L. Heitmeyer, and S. Sims. Tame: A PVS interface to simplify proofs for automata models. In *UIT-P'98*, 1998.

[5] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *ISSTA*, pages 280–292, 1996.

[6] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Soft. Eng.*, 19(1):24–40, January 1993.

[7] G. S. Avrunin, J. C. Corbett, and L. K. Dillon. Analyzing partially-implemented real-time systems. In *ICSE*, pages 228–238. ACM Press, 1997.

[8] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6:37–68, 1999.

[9] G. Booch. *Object Oriented Design*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[10] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

[11] J. P. Bowen and M. J. C. Gordon. Z and HOL. In *Proceedings of the 8th Annual Z User Meeting*, Workshops in Computing. Springer-Verlag, London, 29–30 1994.

[12] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[13] 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*, 98:142–170, June 1992.

[14] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[15] N. Day. A model checker for statecharts. Master's thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1993.

[16] N. A. Day. *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1998.

[17] N. A. Day, M. R. Donat, and J. J. Joyce. Taking the hol out of HOL. Technical Report 00-003, Department of Computer Science, Oregon Graduate Institute, January 2000.

[18] N. A. Day and J. J. Joyce. Symbolic functional evaluation. In *TPHOLs*, number 1690 in LNCS, pages 341–358. Springer, 1999.

[19] N. A. Day, J. J. Joyce, and G. Pelletier. Formalization and analysis of the separation minima for aircraft in the North Atlantic Region. In *Lfm*, pages 35–49. NASA Conference Publication 3356, September 1997.

[20] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, Englewood Cliffs, New Jersey, 1979.

[21] M. Gordon and T. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[22] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

[23] W. Harrison and H. Ossher. Subject-oriented programming. In *OOPSLA*, pages 411–428, 1993.

[24] M. P. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. on Soft. Eng.*, 22(6):363–377, June 1996.

[25] C. Heitmeyer, James Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Soft. Eng.*, 24(11):927–948, November 1998.

[26] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[27] K. Heninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Trans. on Soft. Eng.*, SE-6(1):2–13, January 1980.

[28] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279–295, May 1997.

[29] i-Logix Inc., Burlington, MA. *The Semantics of Statecharts*, January 1991.

[30] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. In *International Workshop on the HOL Theorem Proving System and its Applications*, pages 285–299. Springer, 1994.

[31] N. Leveson, K. Bauer, M. Heimdahl, W. Ohlrich, K. Partidge, V. Rata, and J. Reese. A CAD environment for safety-critical software. University of Washington Safety-Critical Systems Project, July 1995.

[32] N. G. Leveson, M. P. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Trans. on Soft. Eng.*, 20(9):684–707, September 1994.

[33] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[34] G. Murphy, Oct. 1996. Personal Communication.

[35] R. F. Paige. Heterogeneous notations for pure formal method integration. *Formal Aspects of Computing*, 10(E), 1998.

[36] D. L. Parnas. Tabular representations of relations. Technical Report 260, Communications Research Laboratory, Faculty of Engineering, McMaster University, October 1992.

[37] J. L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[38] M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools. In *ICSE*, pages 239–249. ACM Press, 1997.

[39] P. S. Rajan. *Transformations on Data Flow Graphs: Axiomatic Specification and Efficient Mechanical Verification*. PhD thesis, Dept. of Comp. Sci, Univ. of British Columbia, 1995.

[40] Rational Software. *UML Notation Guide: Version 1.3*. Available at "http://www.rational.com/uml".

[41] J. Spivey. *Understanding Z*. Cambridge University Press, Cambridge, 1988.

[42] E. Y. Wang, H. A. Richter, and B. H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *ICSE*, pages 45–55. ACM Press, May 1997.

[43] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.