REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 31-01-2007	**2. REPORT TYPE** SBIR Phase I Final Report	**3. DATES COVERED** *(From - To)* Aug 2006 - Jan 2007

4. TITLE AND SUBTITLE

A Software Hub for High Assurance Model-Driven Development and Analysis

5a. CONTRACT NUMBER

N00014-06-M-0145

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Rance Cleaveland
Steve Sims
David Hansel
Dan DuVarney

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Reactive Systems, Inc.
120-B E. Broad St.
Falls Church VA 22046

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
ONR 251
875 N. Randolph St., Suite 1425
Arlington VA 22203-1995

10. SPONSOR/MONITOR'S ACRONYM(S)

ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

N/A

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited / Unclassified

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This final report describes the results of a Phase I SBIR research project sponsored at Reactive Systems Inc. by the Office of Secretary of Defense, with oversight provided by the Office of Naval Research. The goal of the project was to conduct a feasibility study for a so-called software hub that is intended to promote interoperability among software modeling and analysis tools. As part of this six-month effort a translator was implemented from the commercially popular modeling notations Simulink / Stateflow into the SAL input notation for the SALSA analysis tool, and several experiments conducted that demonstrated the utility of applying SALSA-style analyses to Simulink / Stateflow models. A preliminary design of a software hub was also developed.

15. SUBJECT TERMS

Software hub, software models, model checking and analysis, software verification, software design

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Steve Sims
UU	UU	UU	None	34	**19b. TELEPHONE NUMBER** *(Include area code)* 703-534-6458

A Software Hub for High Assurance Model-Driven Development and Analysis

SBIR Final Report
January 23, 2007

Reactive Systems, inc
www.reactive-systems.com

Topic Number:	OSD05-SP4
Award No.:	N00014-06-M-0145
Contractor Name:	Reactive Systems, Inc.
Contractor Address:	120-B East Broad St. Falls Church, VA 22046

This final report describes the Phase I SBIR research performed at Reactive Systems, Inc. (RSI) under the Office of Naval Research (ONR) sponsored project titled *A Software Hub for High Assurance Model-Driven Development and Analysis.*

Phase I Technical Team

- Rance Cleaveland (CEO)
- Steve Sims (CTO)
- David Hansel (Lead Reactis developer)
- Dan DuVarney (Lead Reactis for C developer)

Introduction

Embedded software applications are rapidly growing both in number and complexity. Developing such software poses significant challenges to software and system engineers. One appealing approach, Model-Based Design (MBD), uses executable models to drive the design, specification, implementation, and validation phases of a system-development project. Growing numbers of system and software engineers are using MBD in military, aerospace, automotive, medical-device and other embedded applications, as it enables the iterative development of precise specifications and designs while at the same time supporting the integral involvement of application experts in the software-development aspects of implementation. Developers are further leveraging their modeling efforts through the use of code generators and model-based testing and validation tools to automate other activities in the system life cycle. A primary, overarching benefit of the MBD approach is that it promotes the early detection of design errors and safety violations.

In the automotive and aerospace industries, the most widely used modeling notation for MBD is the Simulink® /Stateflow®[1] language offered by The MathWorks. Other notations used for MBD include the Unified Modeling Language (UML), MATRIXx, the variant of Statecharts supported by I-Logix, SDL, Esterel, Lustre, and others.

[1] Simulink and Stateflow are registered trademarks of The MathWorks, Inc.

In parallel with the commercial development of modeling languages and simulation tools, the Computer-Science research community has developed an array of formal-verification tools, including model checkers, theorem provers, and consistency checkers, and advanced deployment environments. These tools analyze models at a level of mathematically rigor that traditional, testing-based V&V approaches cannot approach; the level of confidence they can provide in models is therefore much higher. The integration of such analysis tools into MBD processes has the potential to dramatically improve the quality of embedded software applications while at the same reducing the costs associated with deploying such applications. A major obstacle to such an integration is the fact that each verification tool employs a different modeling notation and they typically do not support those notations used in commercial MBD settings.

The work undertaken in this Phase I project aimed investigate the feasibility of a *software hub* as a remedy this lack of interoperability. The envisioned framework, shown in Figure 1, would consist of a general purpose, intermediate specification language that will serve as the means of communication among various tools. The hub will include translators from various commercial modeling notations such as Simulink / Stateflow and UML into the hub language, as well as translators from the hub language to various analysis engines. The hub also offers an opportunity to develop other tool support for enhancing development processes, such efficient code generators that would eliminate the need for manual generation of source code implementations from models and leverage advanced deployment environments. RSI will leverage its Reactis®[2] testing and validation package for Simulink/Stateflow models in the development of the hub.

For this Phase I project, we implemented a prototype translator from a subset of the widely used Simulink / Stateflow notation of The MathWorks into the SCR Abstract Language (SAL), the input notation of the Salsa [BS00] invariant checker.   The design, development, debugging, and optimization of the translator proved to be a very useful vehicle for exploring issues related to the design of a hub language. The successful application of the translator to a test bed of example models and subsequent analysis of the generated output with Salsa indicates that the appropriately designed hub and hub language and might facilitate much greater tool interoperability than currently exists.

[2] Reactis is a registered trademark of Reactive Systems, Inc.

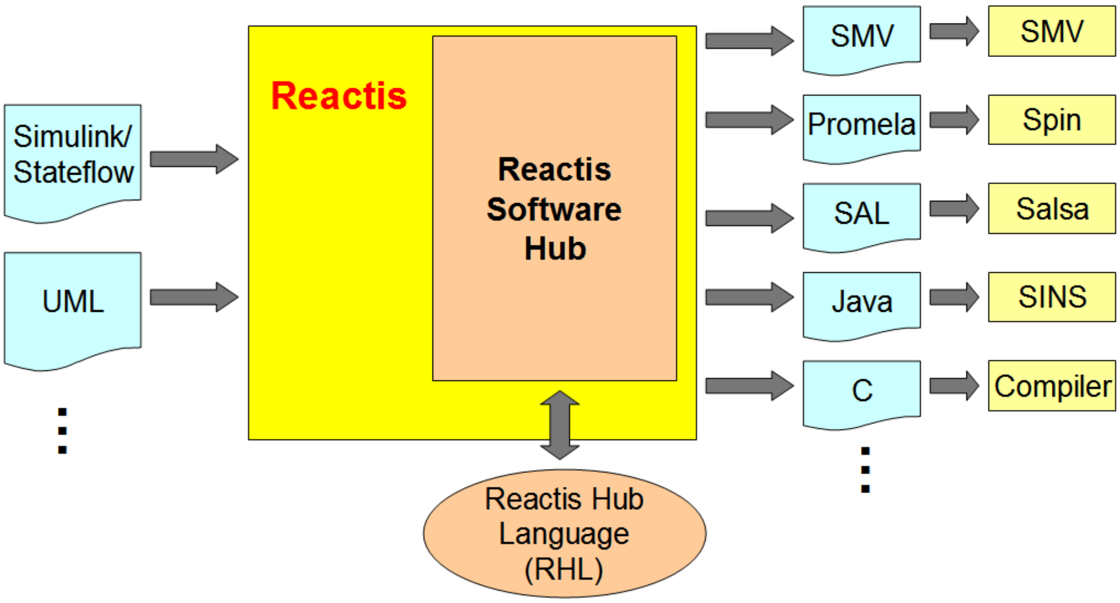


Figure 1: The envisioned software hub framework aims to remedy the lack of interoperability between commercial MBD languages and advanced verification and deployment tools.

Background

Simulink / Stateflow. Simulink / Stateflow is the most widely used modeling language for embedded applications, according to market data collected by the Venture Development Corporation [BL03, LB03]; it is used in many industries, including automotive, aerospace/military, telecommunications, and process control. The Simulink / Stateflow language also shares many features with other modeling languages. For example, the notation includes a facility for defining hierarchical state machines that has much in common with the UML state machine notation. It also includes features for defining architectural and data-flow components of a system that one may find in UML-Real Time and SDL. Thus, while for commercial reasons this project will focus on Simulink/Stateflow, the lessons learned will be applicable to other modeling languages for MBD.

Reactis and MWI. Reactis is a testing and validation environment for Simulink / Stateflow models. The tool was developed by Reactive Systems using SBIR funding from the National Science Foundation; the first release was in 2002. Reactis users can generate test data from Simulink / Stateflow models and also check whether models satisfy user-defined functional requirements. Internally, Reactis works by translating Simulink / Stateflow models into an intermediate notation, MathWorks Intermediate language (MWI), developed by Reactive Systems. The motivation for this notation was efficiency-related; processing it was much faster than repeatedly re-analyzing the Simulink / Stateflow file format. Subsequently, certain RSI customers have developed an interest in MWI as a mechanism for interoperation between the Simulink / Stateflow tool set and model-checking tools. In 2005, a team at the University of Minnesota and Rockwell Collins implemented a translator for converting a subset of MWI into an input format for the popular SMV model checker [BCM+92].

Salsa. Salsa is an invariant checker for models specified in the Software Cost Reduction Abstract Language (SAL), which is very similar to the Secure Operations Language (SOL) [Bha02]. To establish a formula as an invariant, Salsa carries out an induction proof that utilizes tightly integrated decision procedures (currently a combination of binary-decision-diagram algorithms and a constraint solver for integer linear arithmetic) for discharging the verification conditions. Unlike other inductive provers, Salsa works in a totally automatic fashion. Its user interface mimics that of a model checker: given a formula and a model, Salsa either establishes the formula as an invariant of the model or provides a counter example. The use of induction enables Salsa to combat the state-explosion problem that plagues model checkers: it can handle specifications whose state spaces are too large for model checkers to analyze. Also, unlike general-purpose theorem provers, Salsa concentrates on a single task and gains efficiency by employing a set of optimized heuristics.

Implementation of a Prototype Translator

As a vehicle for studying the feasibility of a software hub, we focused on one use case of such a hub, namely the application of Salsa to Simulink/Stateflow models. We anticipate that the lessons learned from this slice of the hub approach will be applicable to both other MBD languages (such as UML) and other back end engines. The scope of the Phase I work is illustrated in Figure 2.

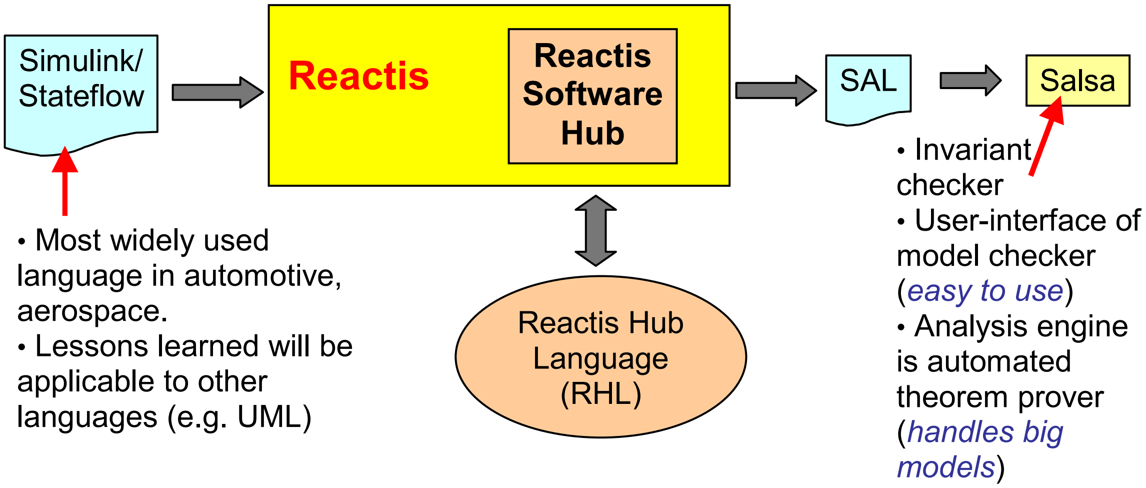

Simulink/ Stateflow
Reactis
Reactis Software Hub
SAL
Salsa
Reactis Hub Language (RHL)
• Most widely used language in automotive, aerospace.
• Lessons learned will be applicable to other languages (e.g. UML)
• Invariant checker
• User-interface of model checker (*easy to use*)
• Analysis engine is automated theorem prover (*handles big models*)

Figure 2: The Phase I project developed a Simulink/Stateflow-to-SAL translator to investigate the feasibility of a hub language as a means of communication among different tools.

Reactis reads an MDL file (the file format in which The MathWorks tool suite stores Simulink/Stateflow models) and then outputs a MWI file storing an intermediate representation of the model. The work undertaken in this Phase I project yielded a translator *mwi2sal* which reads an MWI file, transforms the model encoded in that file, and outputs a SAL file suitable for processing by Salsa. This yields the usage scenario, shown in Figure 3, for applying Salsa to a Simulink/Stateflow model cruise.mdl.

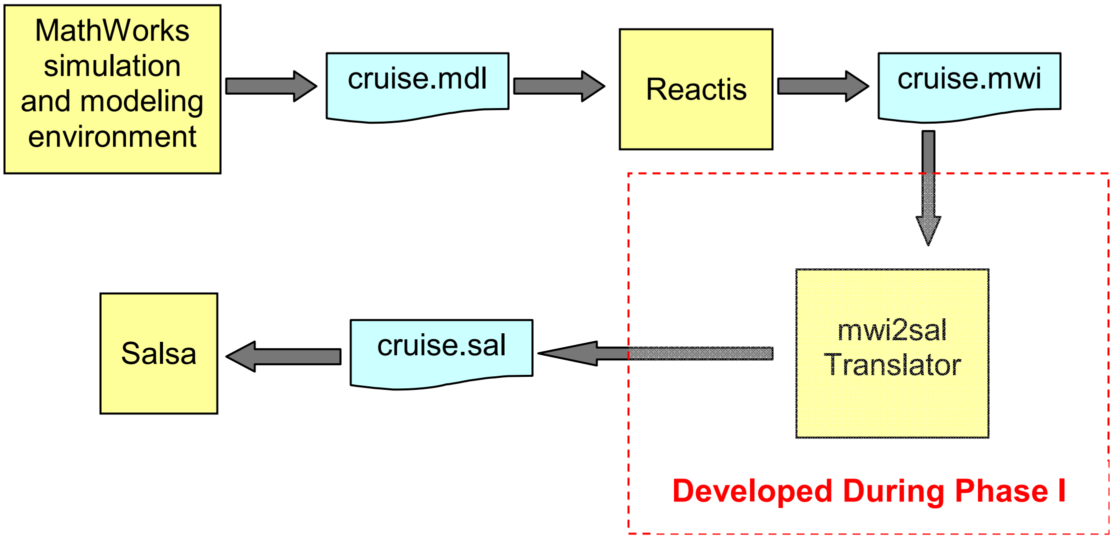


Figure 3: The mwi2sal translator (along with Reactis) provides the means for applying Salsa to Simulink/Stateflow models. A prototype of this technology has been developed during the project.

MWI files include two main sections, one capturing the Simulink portion of a model and one capturing the Stateflow portion. A different translation strategy was required for the two sections. We now discuss some of the issues encountered when developing the translation strategy for each component.
In the following discussion we denote the Simulink portion of MWI as MWI-SL and the Stateflow portion as MWI-SF.

Simulink Section of MWI

Similarities with SAL

A number of similarities between MWI-SL and SAL facilitate translation.

Both languages incorporate hierarchical model decomposition. In MWI-SL a model consists of a set of *subsystems* each of which may contain of a set of child subsystems. SAL includes the same type of model structuring mechanism, although the SAL terminology denotes the components *modules* instead of subsystems. mwi2sal translates each MWI-SL subsystem to a SAL module.
Both MWI-SL subsystems and SAL modules define component behavior using three sets of variables: inports, outports, and local variables in the case of MWI-SL and monitored, controlled, and internal variables in the case of SAL. The two languages also define how

each variable is updated during a simulation step in a similar fashion, namely, via a definition consisting of an expression that specifies how the variable is updated. mwi2sal simply translates variables and definitions from the source language to the corresponding equivalent in the target language.

The expression notations of the two languages have both similarities and differences. Similar constructs include standard arithmetic and logical operators as well as if-then-else statements. Differences are discussed in the following section.

Another way in which MWI-SL and SAL are alike is that they both employ a synchronous semantics. That is a model executes by reading its inputs and then computing a new value for each variable before accepting a new set of inputs. Both languages also compute the variable update order based on the dataflow among variable definitions.

Differences with SAL

While the similarities enumerated above facilitate translation, a number of differences in the languages complicate the task. Some of these have already been addressed in the prototype translator, and others have been deferred until a possible Phase II project. We now discuss some of these differences.

The expression language used to construct variable definitions of MWI-SL has many more operators than that of SAL. Some of these may be transformed via simple operations to SAL primitives. For example, the SAL has no exclusive or operator, therefore *A xor B* is translated to *(A or B) and not (A and B).* Other MWI-SL constructs can simply be discarded by the translator since they have no effect on verification. For example, MWI-SL includes coverage tags that Reactis uses to track which parts of a model have executed. These are not needed by Salsa. Other constructs are more complicated to translate. For example, when the translator encounters an array it generates a SAL variable for each element of the array and transforms all indexing of the array to be references to the appropriate variable.

Some of the other constructs whose implementation in the translator was deferred until Phase II are the following.

- Floating point values. Single and double precision floating point types are by far the most commonly used types in industrial Simulink/Stateflow models. Therefore extension of the translator and Salsa to handle these will be necessary for commercial viability of the software hub. The current Salsa invariant checking technique could be extended to handle real numbers during Phase II.
- Fixed-point types. This technique of representing real numbers using integers along the appropriate scaling is often used when Simulink/Stateflow models will ultimately be transformed into C code that will run on a target microprocessor that lacks a floating point unit. Since all underlying operations are performed on integers, these types could be handled by the existing Salsa along with extensions to mwi2sal.

- Multi-rate models. The Simulink/Stateflow language allows different parts of a model to execute at different time intervals. For example, one part might execute every second while another executes only every 5 seconds. One possible approach to handling multi-rate models would be for mwi2sal to generate the appropriate scheduling information as a part of the SAL it outputs for a model. If such that approach yields models too complex for analysis, an alternative approach would be to analyze the parts of the model that execute at different rates independently.
- Loops. Simulink includes several types of loop blocks and looping behavior can easily be modeled in Stateflow.
- Triggered subsystems. Non-triggered subsystems and SAL modules include variables that are updated according to an ordering inferred from the dataflow of the variable definitions. In contrast MWI-SL triggered subsystems (and all variables contained therein) only executed then a triggering condition is satisfied.
- Global variables.
- Exceptions.
- Math operators. sine, cosine, …

Stateflow Section of MWI

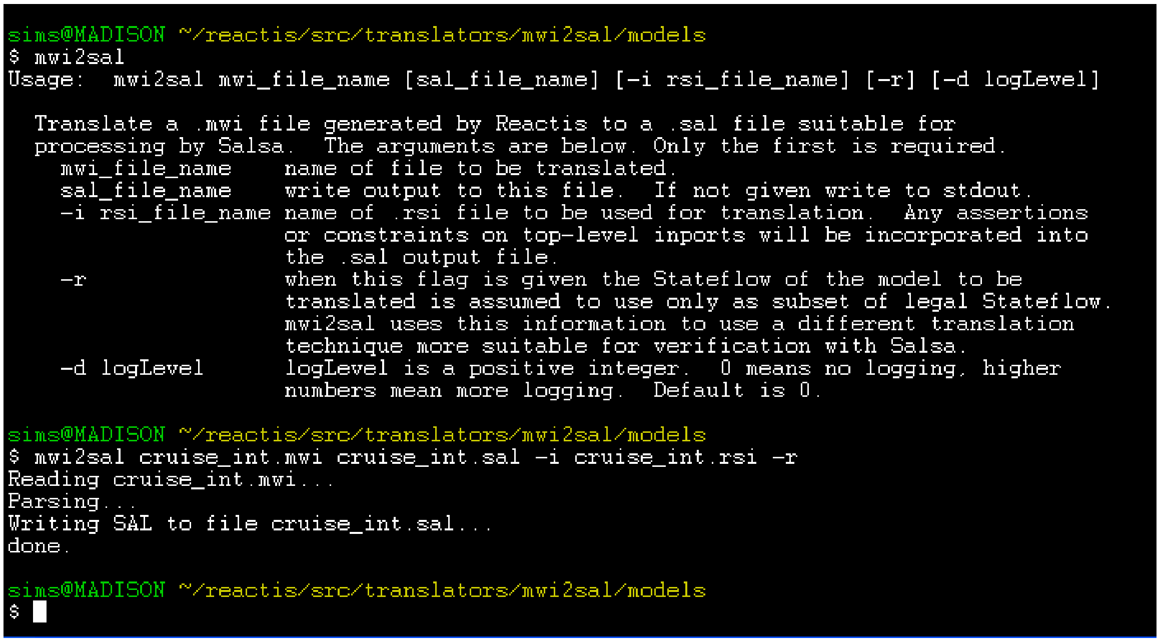
Stateflow charts are a graphical representation of hierarchical finite-state automata extended with variables, functions, event broadcasting, and concurrency. Stateflow charts are based on Statecharts [HP98,Har87]. For more details on Stateflow, see Subsection "Stateflow" of Section "Reactis Hub Language" below. MWI includes a component (denoted MWI-SF here) for capturing the specification of Stateflow diagrams. This portion of MWI is more dissimilar from SAL than the Simulink portion discussed above, thereby complicating the mwi2sal implementation.

The general strategy we followed to translate MWI-SF is as follows. For each state containing child states, a SAL variable of enumerated type is generated. The type includes one value for each child state. The definition of the behavior each state consists of an if statement with a guard corresponding to each transition from or to the state or within the state (between child states). Additionally updates to Stateflow variables are supported when they occur within state entry, during, or exit actions.

Many Stateflow features have been deferred until Phase II for implementation in mwi2sal. These include: condition and transition actions, events, junctions, function states, multiple assignments to variables, local variables in states, temporary variables, non-scalar values. Additionally, output ports must have initial values.

User Interface

The mwi2sal translator currently has a command line interface. The user launches the tool by giving a MWI filename as an argument. If an optional SAL filename argument is also given, then the translated version of the model will be written to that file. If no output file is specified the SAL output is written to standard out. The other supported arguments are described below in Figure 4.



```
sims@MADISON ~/reactis/src/translators/mwi2sal/models
$ mwi2sal
Usage:  mwi2sal mwi_file_name [sal_file_name] [-i rsi_file_name] [-r] [-d logLevel]

  Translate a .mwi file generated by Reactis to a .sal file suitable for
  processing by Salsa.  The arguments are below. Only the first is required.
    mwi_file_name    name of file to be translated.
    sal_file_name    write output to this file.  If not given write to stdout.
    -i rsi_file_name name of .rsi file to be used for translation.  Any assertions
                     or constraints on top-level inports will be incorporated into
                     the .sal output file.
    -r               when this flag is given the Stateflow of the model to be
                     translated is assumed to use only as subset of legal Stateflow.
                     mwi2sal uses this information to use a different translation
                     technique more suitable for verification with Salsa.
    -d logLevel      logLevel is a positive integer.  0 means no logging, higher
                     numbers mean more logging.  Default is 0.

sims@MADISON ~/reactis/src/translators/mwi2sal/models
$ mwi2sal cruise_int.mwi cruise_int.sal -i cruise_int.rsi -r
Reading cruise_int.mwi...
Parsing...
Writing SAL to file cruise_int.sal...
done.

sims@MADISON ~/reactis/src/translators/mwi2sal/models
$ █
```

Figure 4: mwi2sal currently has a command line interface.

Example Models for Testing mwi2sal

The following small test bed of Simulink/Stateflow models was assembled to test the translator.

- auto_pilot_orig. A SAL model of a simple auto pilot developed by Salsa co-creator Ramesh Bharadwaj.
- auto_pilot. For the purpose of testing mwi2sal we reverse-engineered a Simulink/Stateflow model from auto_pilot_orig.
- gasStation. A slightly modified example from the Reactis distribution that models the control software for a gas station pump control system.
- Cruise Control. Three variations of a simple cruise control model developed at RSI and distributed with Reactis. In each case all floating point values were replaced by integer values suitably scaled. One non-linear constraint was removed.
    - cruiseNoPlant. The feedback loop linking the throttle output to speed was removed.
    - cruise_int_sl. A Simulink-only version.
    - cruise_int. Includes both Simulink and Stateflow.

Table 1: The number of BDD variables and atomic integer constraints as reported by Salsa for each example model.

Example	# BDD Variables	# atomic integer constraints
auto_pilot_orig	50	27
auto_pilot	444	123
gasStation	260	145
cruiseNoPlant	186	115
cruise_int_sl	269	200
cruise_int	242	167

Debugging mwi2sal

This section discusses our efforts to ensure correct operation of the translator. The examples described above were used in these efforts.

The first step in the debugging process was to check that the generated SAL code was syntactically correct. This was achieved through by repeatedly running Salsa, inspecting parse errors and correcting mwi2sal.

After removing syntax errors, we turned our attention to validating that the SAL output was semantically equivalent to the Simulink/Stateflow model from which it was generated. For this task we extended Salsim (the SAL simulator) to execute test suites generated by Reactis and exported as CSV (comma separated value) files. Using this capability we did the following steps for each example in the test bed.

1. Use Reactis to generate test suite foo.rst (as a byproduct foo.mwi is created)
2. Export foo.rst as foo.csv
3. mwi2sal foo.mwi foo.sal
4. Use Salsim to execute tests in foo.csv on foo.sal. Any differences between the outputs produced by SAL and those produced by the Simulink/Stateflow model are flagged.
5. Based on any differences correct mwi2sal

Optimizing the Tool Chain

After debugging mwi2sal, we had the tool chain in place to apply Salsa to Simulink/Stateflow models. Unfortunately, our initial Salsa runs were not completely successful. Many checks took excessive amounts of time to run, while others never completed due to exhausted memory. In this section we describe the steps we took to improve performance.

Optimization 1: Include User Constraints on Top-Level Inports

Reactis includes a facility that allows users to limit the set of values that may arrive at a top-level input port during simulation. A simple type editor allows for the specification of ranges, enumerated sets of values, and limits on how much a value may change from one step to the next. SAL also includes a similar mechanism for specifying environmental assumptions. Extending mwi2sal to translate these constraints specified within Reactis was straightforward and led to improved performance of one the test models.

Optimization 2: Precompute Infeasible Integer Constraints

Salsa offers a –f flag that causes the tool to precompute and cache sets of infeasible integer constraints. The flag accepts a single argument that may be 0, 2, or 3 to indicate the size of the collections of constraints to be checked for feasibility during a preprocessing phase. We found that changing this flag from the default value of 0 to 2 worked best for the examples we tested. This configuration causes Salsa to do the following prior to starting an invariant check.
For each pair of atomic integer constraints *A* and *B* in the model, check whether *A and B* is satisfiable; if not, any conjunction Salsa tries to solve during the subsequent invariant check that includes these constraints will be known to be unsatisfiable (and therefore the conjunction need not be checked for satisfiability).

Optimization 3: Alias Elimination

By studying the SAL output of mwi2sal as well as the output statistics of Salsa we noted that the translator was generating a substantial number of variable aliases (variable definitions of the form "var x = y*"*). While aliases may seem harmless, they cause Salsa to generate extra BDD variables and integer constraints that cause its invariant checking engine to get bogged down and perform poorly.

Our first idea for resolving this issue was to modify mwi2sal to produce fewer aliases. Unfortunately, we determined that this approach would not address the bulk of the aliases produced by mwi2sal. The reason for this was that, because of the way MWI "wires together" different subsystems, many aliases have the form x = B.y (where y is an output of subsystem B). In other words the aliases cross subsystem boundaries. Since mwi2sal generates a SAL module for each Simulink subsystem, it is not possible to eliminate these cross-module aliases during translation. One alternative considered was to have mwi2sal flatten the Simulink model and generate a single SAL module. We rejected this approach because we predict the utility of being able to perform Salsa checks on all modules within a model instead of only the top-level module.

The approach we took for removing aliases was to modify Salsa itself to perform this task. This enhancement required a substantial programming effort. Since Salsa combines modules as it instantiates the model, a simple syntactic transformation was not feasible (for the same reason it could not be done in mwi2sal, namely cross-module aliases). Instead it was necessary to implement a transformation of intermediate data structures to eliminate the aliases.

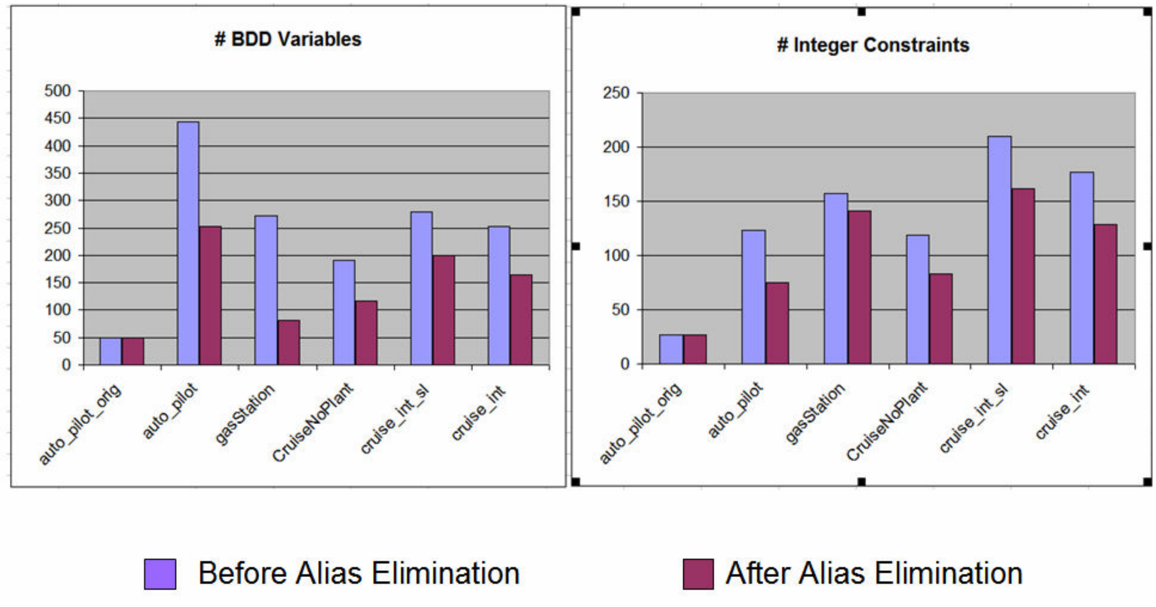


Figure 5: The alias elimination optimization eliminates causes a decrease in both BDD variables and integer constraints.

As can be seen in the data below, this optimization yielded the largest benefits by far (approaching factor 10 improvement in several cases).

Optimization 4: Port Salsa to New Compiler

The final optimization that we describe was to port Salsa to the MLton Standard ML compiler (www.mlton.org). This relatively new compiler takes a novel approach to compiling functional programs, namely *whole program optimization*.
Our experiments noted a roughly factor 3 improvement in runtime performance by porting from the previously used SML/NJ compiler to MLton. In addition to performance, MLton has several additional advantages. It generates standalone executables that eliminate the need for launch scripts that can be a challenge to maintain on the Windows platform. Salsa was originally developed on a Unix/Linux platform where such scripts are less problematic. The porting effort required some slight reorganizations of the code base, some updates to reflect changes in libraries, the elimination of non-standard features used by the SML/NJ compiler, and the development of a set of .mlb files that define how a program should be compiled by MLton. After the successful port, Salsa now compiles and runs under both Linux and the Cygwin environment on Windows.

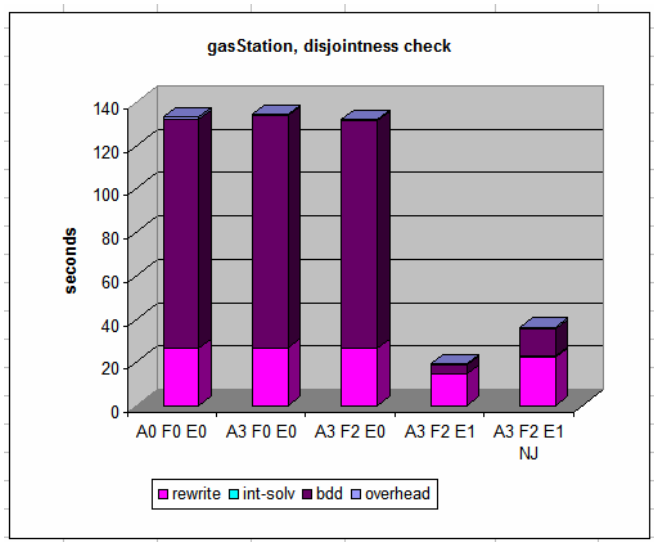
Performance Statistics

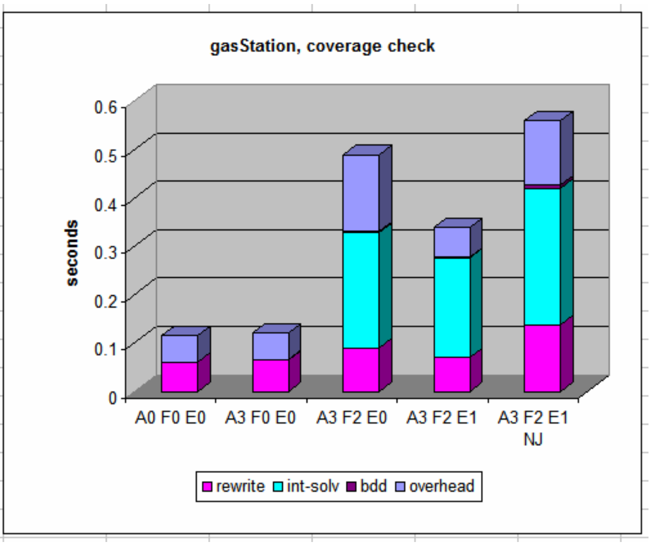
To measure the effectiveness of our optimization strategies we performed a set of Salsa runs. Below are graphs showing the performance figures. Results of performing consistency checks for *disjointness* (no nondeterminism) and *coverage* (no missing cases) are shown for each example model. In the graphs, each bar shows the run time for Salsa to form the check indicated in the title. The different partitions within each bar shows the breakdown of the different tasks associated with the check:

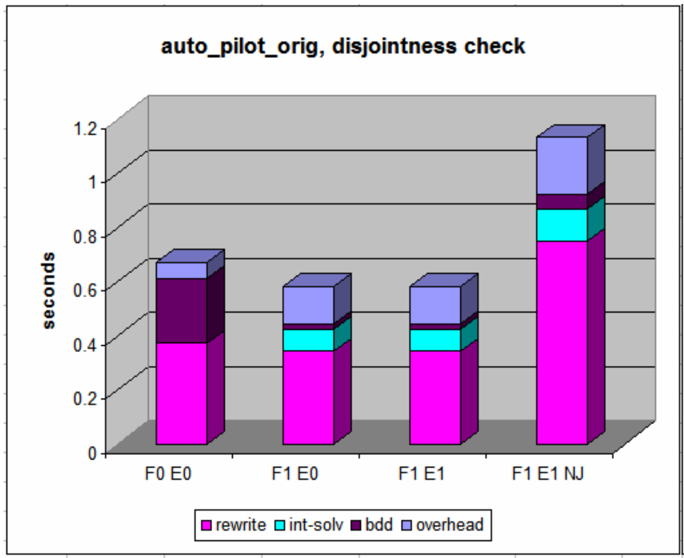
- rewrite – time transforming SAL expressions into the internal representations used by the invariant checker.
- int-solv – time solving sets of integer constraints.
- bdd – time performing binary decision diagram (BDD) operations.
- overhead – time performing all other tasks.

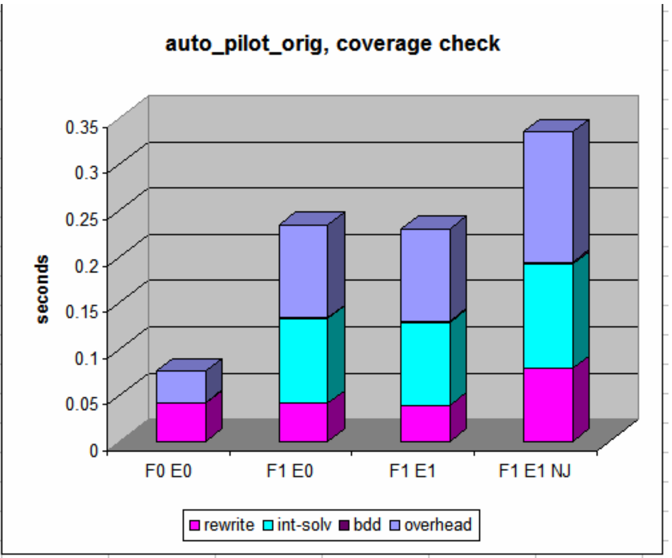
Each bar within a graph represents a run using a different set of the optimizations described above. The leftmost bar shows the case when no optimizations are used. Each bar is labeled by a set of 2-character tags that indicate which optimizations were used.

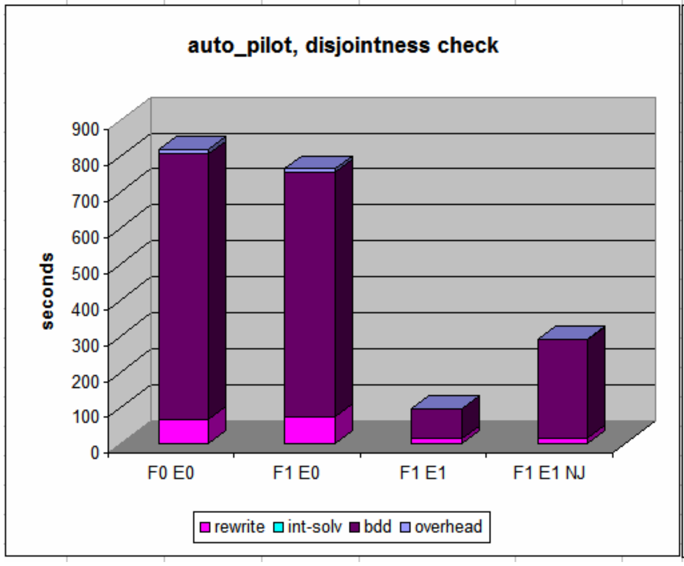
- The tag beginning with A indicates the number of constraints for top-level input ports that were used.
- The tag beginning with F indicates which –f argument was used.
- The tag beginning with E indicates whether aliases were eliminated. Zero means no, one means yes.
- The NJ tag indicates that the SML/NJ compiler was used to compile Salsa. When NJ is not present, MLton was used.

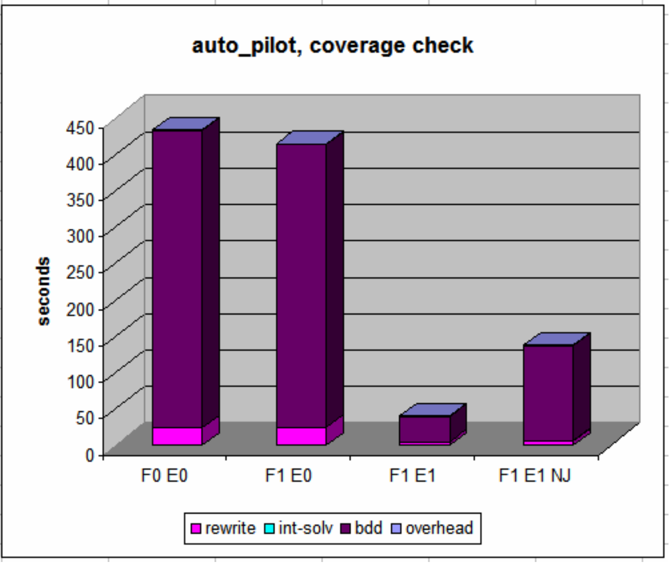


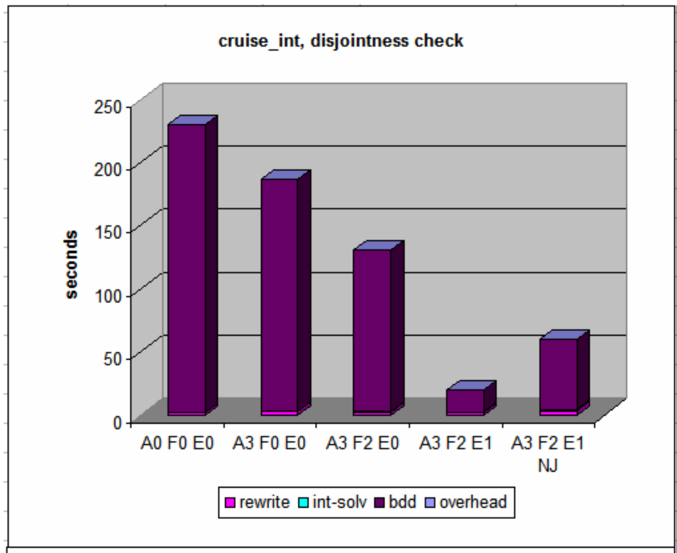


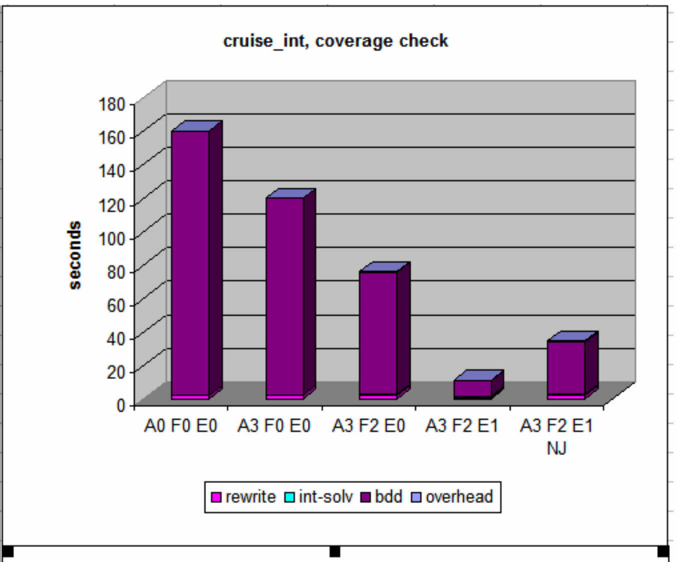

auto_pilot_orig, disjointness check
seconds
1.2
1
0.8
0.6
0.4
0.2
0
F0 E0
F1 E0
F1 E1
F1 E1 NJ
■ rewrite ■ int-solv ■ bdd ■ overhead

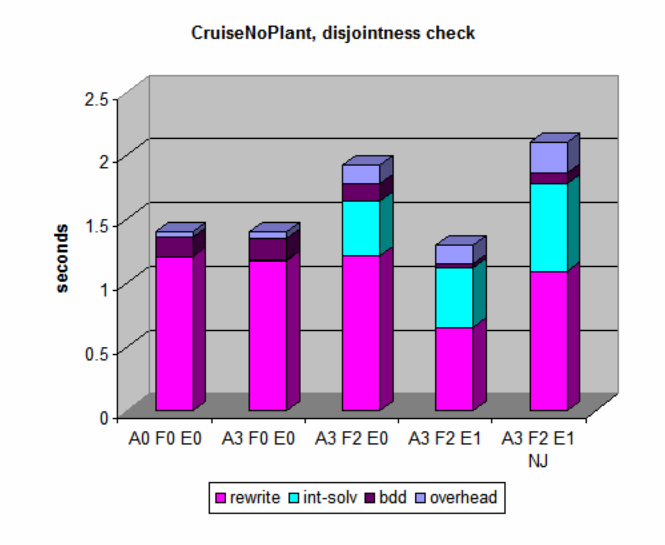

auto_pilot_orig, coverage check
seconds
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
F0 E0
F1 E0
F1 E1
F1 E1 NJ
■ rewrite ■ int-solv ■ bdd ■ overhead

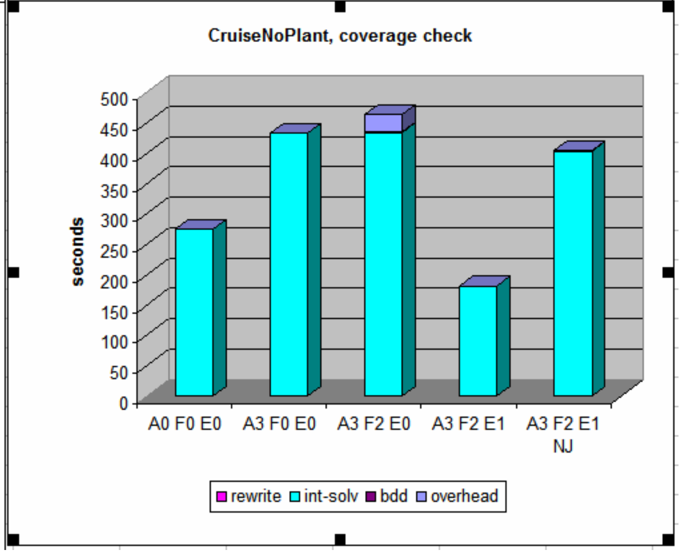

auto_pilot, disjointness check
seconds
900
800
700
600
500
400
300
200
100
0
F0 E0
F1 E0
F1 E1
F1 E1 NJ
■ rewrite ■ int-solv ■ bdd ■ overhead


auto_pilot, coverage check
seconds
450
400
350
300
250
200
150
100
50
0
F0 E0
F1 E0
F1 E1
F1 E1 NJ
■ rewrite ■ int-solv ■ bdd ■ overhead









Reactis Hub Language

Our experience building and tuning the mwi2sal translator was a useful vehicle for investigating the feasibility of developing a software hub to offer better interoperability between commercial MBD tools and advanced verification and deployment tools. Based on those experiences we have also explored many of the issues related to the design of the *Reactis Hub Language (RHL)*, an intermediate representation whose purpose is to represent the properties of Simulink/Stateflow and UML models which are needed for verification and validation by both Salsa and Reactis. In this section, an overview of the two input languages and output languages is presented, followed by the design of RHL.

Simulink/Stateflow

Simulink and Stateflow are integrated modeling languages developed by The MathWorks. Simulink and Stateflow are aimed towards the design and simulation of dynamic systems, such as embedded control systems. This section contains a brief overview of Simulink and Stateflow. For complete details, see [Mat06a,Mat06b].

Simulink

Simulink is a graphical modeling and simulation language. A Simulink model is graphically represented by a *block diagram*, which is composed of a hierarchical collection of *blocks* interconnected by signal-carrying directional arcs called *lines*. Blocks are typically computational units through which data flows, entering from a set of fixed input ports and exiting from one or more output ports (every model must contain at least one signal-generating *source block* and signal-terminating *sink block* in order to be useful). In addition to the input signals, blocks can use previously stored values (called the *state* of the block) to compute the next output. A broad variety of data types are available for stored values, including numeric types, strings, arrays, and records. Signals, on the other hand, are restricted to numeric types only. A typical Simulink model is displayed in Figure 5.

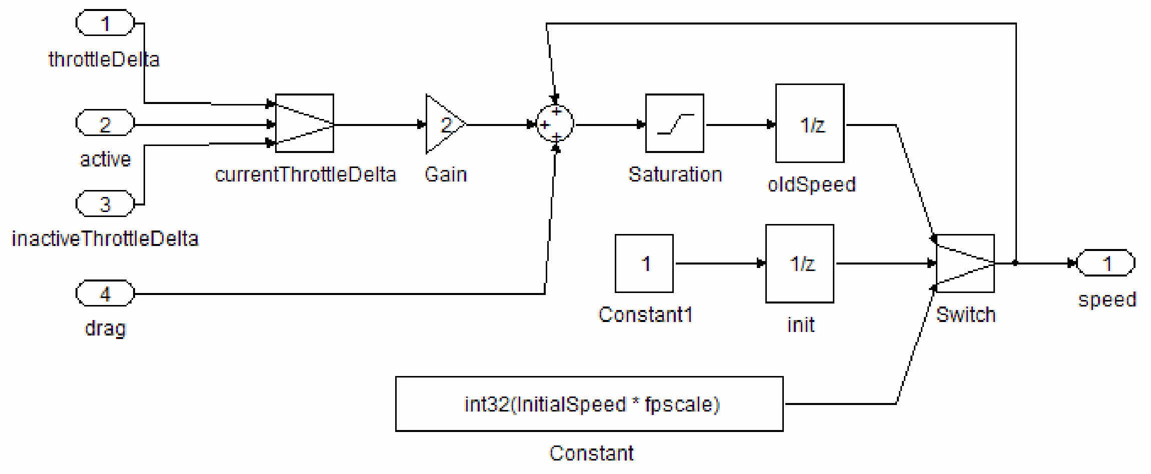


Figure 5: A Simulink Model.

At the lowest abstraction level of a block diagram are *atomic blocks*, which perform mathematical operations such as integer arithmetic, Boolean functions and signal routing. More complex *custom blocks* can be built by interconnecting previously-defined blocks with input and output ports and encapsulating the result. Custom blocks may also be created by encapsulating Stateflow charts, which perform computations using an extended finite state machine paradigm, or by encapsulating C code, which is typically used to improve simulation performance.

Simulink supports both discrete- and continuous-state models. Each block has either no state (e.g., an add block), a discrete state (e.g., a counter block) or a continuous state (e.g., an integration block). Models containing one or more continuous-state blocks can be treated as systems of differential equations and solved using an iterative numerical approximation method such as the 4th-order Runge-Kutta method. Systems consisting of only discrete-state blocks are iteratively simulated by time sampling. The user specifies a *sample time* for each block, and then the sample time for a model is the greatest common divisor of the block sample times. A simulation can either use *fixed-size steps*, in which case the model's outputs are computed for every multiple of the sample time, or *variable-sized steps* can be used, in which case the simulator pre-determines a set of multiples of the sample time when the model generates new outputs, and then computes the model outputs for those times only.

Discrete Simulink simulations have a detailed, albeit informally-defined semantics. Before simulation begins, the model undergoes static analysis and compilation. First, the types and other attributes of all blocks and signals are determined, and some optimizations are performed. Next, the model is *flattened*. Flattening eliminates virtual subsystems by replacing them with their internal components, until the model consists entirely of triggered subsystems and atomic blocks. After flattening, the execution order (called the *sorted order* in Simulink terminology) of the remaining blocks is determined. The sorted order is based on the data-flow dependencies between blocks, so that if an output of block X is connected to an input of block Y, then X appears before Y in the sorted order. In the case of a loop, runtime algebraic methods are used at each simulation step to determine the output values for the blocks in the loop.

The Simulink documentation describes the semantics informally; ambiguities are resolved by consulting the Mathworks' Simulink implementation, which serves as a reference interpreter.

Stateflow

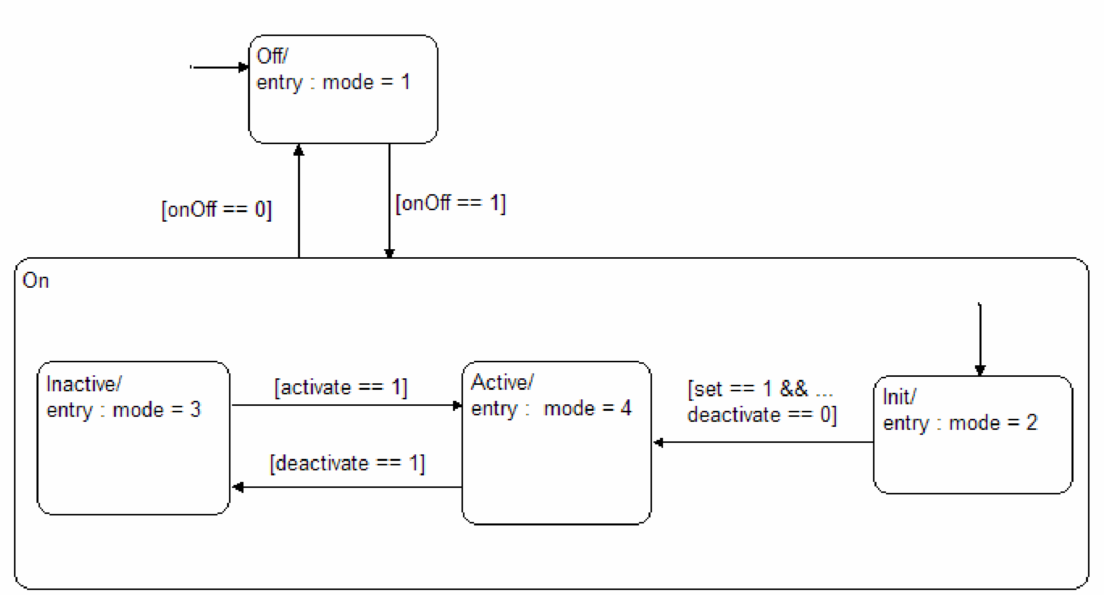
Stateflow charts are a graphical representation of hierarchical finite-state automata extended with variables, functions, event broadcasting, and concurrency. Stateflow charts are based on Statecharts [HP98,Har87]. Figure 6 shows a typical Stateflow chart. The primary components of a Stateflow chart are *states* (the rectangles with rounded edges in Fig. 1) which represent the possible execution points of a Stateflow chart, and *transitions* (the directed arcs in Fig. 1), which control changes in state.

States may contain one or more substates. States are labeled with a name (unique among all the immediate sibling states) plus a set of *actions* to perform. The *initial state* of a collection of states is distinguished by a transition with no source which terminates at the initial state. When a state is activated, its initial state is also activated, except under two conditions. (1) If the incoming transition terminates at or within a substate X, then X is activated regardless of the initial state. (2) If the state is annotated with a *history junction*, the most recently active substate is activated instead of the initial state.

State actions are grouped into five categories. *Entry actions* are performed when a state first becomes active. During actions performed after each global clock tick that the state remains active. *Exit actions* are performed when a state a deactivated. *On actions* define guarded actions to be taken when the state is active and a named *event* is broadcast. So-called *bind actions* aren't really actions, but instead list variables which can only be modified within the state, and events which can only be broadcast by the state. The set of possible actions include assignments, complex expressions (including function calls), and event broadcasts.

A *transition* is a graph composed from of one or more directed arcs called *transition segments*. Transition segments either flow into a *connective junction*, which will have one or more outgoing transition segments, or they may flow into a state, which terminates the transition. In addition, each transition segment can have a label which consists of a *guard* and an *action*. Connective junctions are used to modify the flow of control between states, and can act as merge hubs, allowing multiple incoming transitions to share the same ending transition segments. Connective junctions can also be used to split incoming transition segments into multiple outgoing transition segments, in which case the junction acts as a conditional branch[3]. Note that connective junctions are not strictly necessary, but serve the purpose of making charts easier to understand by increasing the

Figure 6: A Stateflow chart.



[3] Sta
transition in cases where multiple transitions are simultaneously possible.

visibility of branching and by allowing common transition behaviors to be captured.

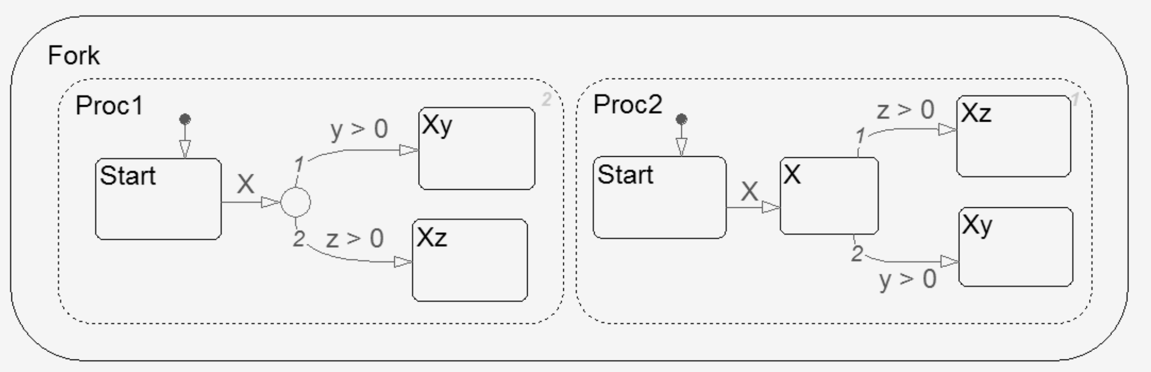


Figure 7: Stateflow chart with and-states and transition junctions.

Transition segment labels consist of an *event trigger*, a *condition*, a *condition action*, and a *transition action*. The (optional) event trigger gives the name of one or more events which must occur (i.e., be broadcast by some state). The condition is a Boolean expression which must be true. Together, the event trigger and the condition act as guard on the transition. Both must be simultaneously true for the transition segment to be enabled. The transition action is an action which occurs whenever the transition is taken, while the condition action is an action which is taken whenever the condition is evaluated to true regardless of whether or not the transition is actually taken.

Concurrency is supported in Stateflow via a special type of state called an *and-state*. An and-state effectively defines a sub-process of its parent state. When a state directly containing and-states is activated, all the and-states become simultaneously active. Each and-state will typically contain a sequential sub-state machine. These sub-machines then execute in a limited form of parallelism, where the order of execution between the and-states is deterministically defined by the Stateflow semantics. Figure 7 shows an and-state with two parallel sub-states (the dashed border indicates an and-state). In Stateflow terminology, a “regular” state (i.e., one with substates that execute sequentially) is called an *or-state* (e.g., in Figure 7, Proc1 and Proc1 are or-states).

Stateflow events may represent inputs from the external environment (which is probably a Simulink model), or may be generated by actions. An action within an and-state can generate a *broadcast event*, which is sent to all siblings of the and-state (i.e., all the processes which started at the same time as the current processes), or it may produce a *directed event*, which is sent to a named state.

If the state which receives an event is not active, nothing happens. If the state is active, the state is first checked to see if it triggers a transition exiting directly from the state. If such a transition is found, the current state is exited (and any exit actions are executed), after which the actions associated with the transition are executed, and finally the destination state of the transition is entered. Determining whether a transition is eligible to be executed is complicated because of the possibility of junctions. The Stateflow rules specify a canonical order for searching the digraph of transition segments to find a state-to-state transition path. The search is short-circuited, i.e., the first such path found is taken, and other possible paths ignored. This eliminates non-determinism in cases where multiple transitions are simultaneously possible.

A final variant of event is the *function call event*. Function call events are similar to a function call with no return value (although the return value(s) can be passed via other means). The Simulink simulator allows recursive function call events, although other Simulink tools, such as the HDL code generator, reject recursive models.

This concludes the overview of Stateflow/Simulink. It should be emphasized once again that many details were omitted for the sake of brevity. See the reference manuals [Mat06a, Mat06b] for the full details.

UML

UML (Unified Modeling Language) is a modeling language designed to visually capture the design of a software system. UML provides the system designer with a dozen or so diagrams, each of which is used to reflect a particular aspect of a system's design. UML diagrams can be partitioned into four main areas (structural, dynamic, physical, and model-management). Of these, only the first two are particularly relevant to the design of RHL. In the following sections, an overview of the structural and physical UML components is presented. A much more detailed presentation can be found in the UML reference manual [RJB04].

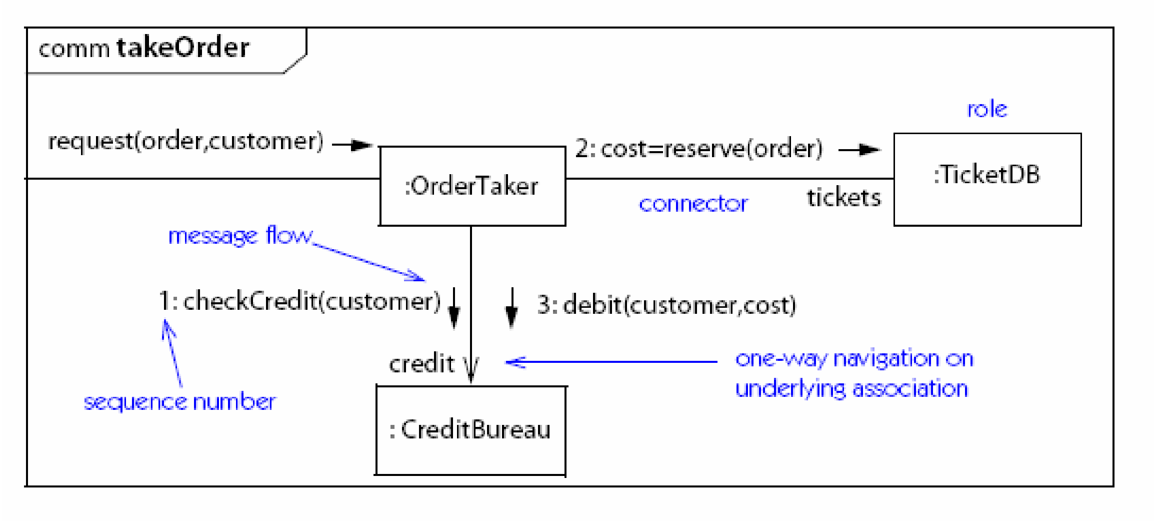


Figure 8. A UML communications diagram (from UML Reference Manual, pg. 107)

Object Orientation

UML is intended to support object oriented design (OOD). The essential features required by OOD are *objects*, *classes*, *inheritance*, and *dynamic message dispatching*. An *object* encapsulates state. Objects are similar to records whose fields cannot be directly accessed, but are instead hidden behind an interface of function which manipulate the field values (in OO terminology, these functions are called *methods*). A method is invoked in response to a message received from another object. The method may do any or all of the following: modify the state of the object, send messages to other objects, or compute a value (in classic OO, all values are objects) that is returned to the calling object.

A *class* is similar to a type. It provides one or more ways to construct objects of the class, what methods are available for those objects, and the type signatures of the methods. Classes are organized into a tree, typically with a universal class such as *Object* at the root. When a class is a child of another class within this tree, it is said to be a *subclass* of its ancestor classes (which are called *superclasses*). A subclass *inherits* from its superclass, which means that any method defined in the superclass is also defined in the subclass. The subclass may modify the behavior of any inherited methods, but the type signature must remain compatible with the type of the method defined in the superclass.

The power of inheritance comes from the combination of *sub-typing* and *dynamic message dispatching*. In any OO system, all subclasses of a class are also subtypes of the class. For example, assume that a class *Figure* has two subclasses, *Circle* and *Square*. *Figure* defines a method *draw(x, y)*, which draws a copy of itself at location *(x ,y)*. A designer building a *Screen* class need not know anything about *Circle* or *Square* in order to display *Figures* on a *Screen* object. The designer can simply maintain a collection of *Figures* and send the draw message to each *Figure* in the container in order to paint the entire *Screen*. Since any subclass of *Figure* is also a subtype of *Figure*, both *Circle* objects and *Square* objects, as well as any instances of any other *Figure* subclass can be stored within a *Screen* object without modifying the code. When the message *draw* is sent to a *Figure* object, the message is dynamically dispatched to the deepest implementation of draw within the class tree, i.e., if the *Figure* object was originally constructed as a *Square* and then cast to type *Figure*, then the method *Square.draw* will be invoked, and not the method *Figure.draw*. Dynamic dispatching is commonly implemented by adding an extra pointer to each object which points to an array of pointers to functions which implement the methods defined for the object.

Structural Diagrams

UML structural views capture the static structure of a software system. The static views are particularly oriented towards object-oriented designs, although non-object-oriented systems can also be represented. There are five standard structural diagrams provided by UML. These are: (a) *class diagrams*, which depict relationships such as inheritance between classes, (b) *internal structure diagrams*, which depict internal components of a class and the interfaces provided-by and required-by the class, (c) *collaboration diagrams*, which depict the interactions between objects that occur when a particular service is performed by the system, (d) *component diagrams*, which show the internal structure of system modules in a manner similar to class diagrams, and (e) *use case diagrams*, which include both system components and external agents and depict communication patterns during a specific usage of the system.

Dynamic Diagrams

UML Dynamic diagrams capture the runtime behavior of a software system. The dynamic diagrams are: (a) *state machine diagrams*, which specify the behavior within objects using a notation similar to Stateflow, (b) *activity diagrams*, which depict communication and concurrency between the activities performed by a system using a notation reminiscent of Petri-nets, (c) *sequence diagrams*, which show communication patterns between potentially concurrent objects along a time axis and (in cases where asynchronous messages are sent) impose a partial order on the actions performed by the objects involved, and (d) *communication diagrams*, which depict communication patterns between objects without a time axis, instead relying upon enumerated tags to indicate the order of message passing. Communication diagrams are the closest analog to Simulink models in the UML.

UML communications diagrams have several components. The first are *objects*, which appear as boxes labeled with the class-name of the object. The second are *connectors*, which represent potential paths for inter-object communications. Third are specific communications which follow connectors in a specific direction. There are three types of communications: *asynchronous messages*, *synchronous calls*, and *return messages*.
A typical communications diagram appears in Figure 4. Calls and messages are labeled with a sequence number (to indicate temporal ordering), a name, and parameter values.

State Machine Diagrams

UML state machine diagrams are quite similar to Stateflow charts, consisting of states, and transitions. As in Stateflow, UML states are hierarchical. UML uses the term *simple states* to indicate a state with no sub-states, and *composite state* to indicate a state with sub-states. A state with sub-states that execute sequentially is called a *nonorthogonal state*. Nonorthogonal states are similar to Stateflow or-states. An *orthogonal state* defines two or more sub-state machines which execute in parallel. UML Orthogonal states are similar to Stateflow and-states, with one significant difference being that the order of execution between simultaneously enabled parallel states is not defined. A UML *Initial state* is the same as a Stateflow initial state. In addition to the initial state, UML also provides a *final state*, which terminates execution of the parent state (i.e., the state immediately enclosing the final state). Stateflow sub-charts, on the other hand, have no final state, but instead execute until a transition is taken which exits the parent state. UML also provides *terminate states*, which stops execution of all parent state machines. Terminate states have no direct equivalent in Stateflow, although they can be simulated by broadcasting an event which is recognized by all states as a halt command. It should also be noted that in single-threaded Stateflow (sub-) charts, a transition exiting the outermost parent is equivalent to a terminate state.

UML also provides *junction pseudostates*, which are very similar to Stateflow junctions. In addition, UML state machines may also contain *choice pseudostates*. A choice pseudostate is essentially a specialized junction which represents a case statement. The choice junction has multiple guarded outgoing transition segments, at least one of which must be true every time the junction is activated (an *[else]* guard can be used to guarantee this).

UML also provides a *shallow history pseudostate*, which is identical to the Stateflow history junction. In addition, UML provides a second type of history pseudostate, the *deep history pseudostate*. Deep history means that each time the enclosing compound state is exited, the last state is pushed onto the stack and upon the next entry to the state, the initial state is determined by the state on top of the history stack. The history stack of the compound state is not popped until the enclosed state sub-machine reaches a final state, at which point one state is popped from the history stack. Deep history states provide support for recursion. In Stateflow/Simulink recursion is not possible because there is no call/return mechanism between blocks.

In addition to orthogonal states, UML state machines may use *fork* and *join* junctions to create and manage concurrency. Fork and join junctions are represented by bars (similar to Petri-nets). A bar with one incoming transition arc and multiple outgoing arcs is a fork, while an arc with multiple incoming arcs and one outgoing arc is a join. When a fork transition is taken, all the target states are simultaneously activated. A join junction acts in the opposite manner, by blocking until all the source states are active and any guards are satisfied at which point the target state is activated. Joins are used to synchronize activity by waiting until two or more sub-processes complete before proceeding. Forks and joins are generalizations of the orthogonal state concept, with one major difference being that a fork need not ever be matched with a join, in which case a model may spawn an unbounded number of processes, resulting in an infinite-state system.

UML provides four kinds of event. A *call event* represents a synchronous call from an external object with parameters the parameters values (if any) are bound to variables specified by the receiver of the call event. Call events are similar to Stateflow function call events. A *signal even*t is similar to a call event, but is asynchronous. A *time event* occurs at a specified point in time, which may be either absolute or relative. A *change eve*nt is a trigger which occurs when a given expression is satisfied. The event triggers immediately at the point when the expression changes from unsatisfied to satisfied (e.g., if the expression is $x > 0$, and the initial value of x is -1, the first assignment of a non-negative value to x will trigger the event). Signal events can be used to create systems which are very difficult to understand, as they create implicit and potentially very complex communication patterns.

RHL Requirements

RHL is intended to serve as an intermediate representation which can read models expressed in either UML or Simulink/Stateflow , and translate them into forms which can be processed by Reactis and Salsa as well as other tools in the future. Hence, RHL must provide a set of features which are sufficiently broad to support the essential components of both UML and Simulink/Stateflow while avoiding constructs which produce intractable behavior in verification tools (e.g., unbounded state growth).

A fundamental basic required feature is support for Statechart-like finite automata extended with mutable variables which can hold values of standard types such as numeric types, arrays, records and textual strings. These EFAs must also have the ability to communicate with each other and their external environment via messages.

Additionally, some sort of encapsulation mechanism (or mechanisms) is required to represent UML objects and Simulink blocks. The encapsulation will need to provide interfaces and communication methods capable of supporting function calls, method dispatching, and data flow interfaces.

There are also a number of common language features which are problematic and which should be restricted in RHL. First is dynamic memory allocation. Dynamically allocated data structures pose extreme difficulties for verification, as they can introduce infinite states and require the verification system to handle pointers. Furthermore, guidelines for embedded control systems (the primary target of RHL) commonly recommend against the use of dynamic allocation, in part to avoid unpredictable real-time behavior due to garbage collection. Hence, this restriction should not be too burdensome.

Function calls are another challenge. Arbitrary recursion can lead to unbounded growth in the state space and make verification intractable. To avoid this, RHL will allow tail recursion only. This will have the benefit of making it possible to flatten models by inlining all function calls. In the experience of the author, embedded control systems rarely employ recursion, so restricting its usage is reasonable.

The use of concurrency can also result in systems which are difficult to verify. Programs which dynamically spawn new processes have unbounded state growth. Even if process spawning is bounded, state space growth is exponential in the number of asynchronous processes. To avoid these problems, RHL will restrict concurrency to within orthogonal states (i.e., Stateflow or-states). This will constrain the amount of state explosion, particularly for Stateflow, which imposes a total ordering on the execution of simultaneously enabled transitions.

Restrictions on object behavior result primarily from the need to avoid dynamic memory allocation. RHL will require all objects to be created at the start of program execution. Classes and inheritance will be supported, and the passing of objects of parameters will be allowed. RHL will allow pass-by-reference of parameters for objects only. This will provide a useable subset of OO features while allowing pointers to be eliminated by cloning functions for each different combination of objects passed as a parameter and then eliminating the parameter via partial evaluation. This restriction will require model designers to limit the number of times that objects are passed as parameters.

RHL will not provide any pointer mechanisms beyond the passing of arguments by reference. There will not be an explicit pointer type. Since dynamic memory allocation is already prohibited, there should be much less need for pointers in RHL models.

A final restriction is that only discrete states will be supported. Continuous states are not possible in digital systems, so this is a quite reasonable restriction. Along the same lines, instantaneous feedback loops (feedback loops without a delay) are not allowed, as such loops are only used to construct models of analog systems.

RHL Overview

Since RHL will be machine-generated, no purely textual syntax is required. The RHL syntactic components are instead defined as nodes in an abstract syntax tree (AST). Each AST node is essentially a variant record whose fields contain the attributes of the node. Every node has two essential attributes. First is the identifier of the node. All identifiers are assumed to be unique. Since RHL is generated from a previously parsed program, it's reasonable to require that all textual identifiers have already been bound and replaced with unique identifiers, so that there are no aliasing or scoping issues.

A second attribute common to all nodes is the *tag* of the node. The tag expresses the basic variety of the node, which is further refined by any additional attributes attached to the node. Some of the attributes presented may seem redundant. To eliminate redundancy, some attributes may be made *virtual* by deriving the attribute value from from other attributes upon request. We now present the set of AST nodes, organized by tag.

Model

Models are the top-level RHL construct. Attributes defined on Models are:

- Globals – a list of all global variables defined within the model.
- Blocks – a list of all simulation blocks defined within the model.
- Types – a list of types defined within the model.
- Classes – the class hierarchy (i.e., the root of the inheritance tree).

Block

Blocks represent functional blocks in a data-flow computation. When invoked during a simulation, a block reads from a set of variables that contain the inputs to the block, performs some computation (driven by a state machine), and then writes its outputs to another set of variables, which will be used as inputs to other blocks. Attributes defined on Blocks are:

- State – The state machine which controls the computation performed by the block.
- Inputs – A list of variables representing input lines.
- Outputs – A list of variables representing output lines.

To facilitate integration with UML, Blocks are also considered to be objects which can be invoked via a method *invoke()*.

Line

Simulink lines are implemented as variables. Line variables are restricted to numeric types as required by Simulink.

State

A state can either represent a single atomic state, or an entire state machine. The attributes of states are:

- SubStates – A list of sub-states, if this state encapsulates a sub-state machine. Atomic states are recognized by an empty *SubStates* attribute.
- InitialSubStates – A list of initial sub-states. If there is only one initial state, then the sub-state machine is sequential. If there are two or more initial states, then the sub-state machine is a parallel machine.
- InTrans – A list of incoming transitions.
- OutTrans – An ordered list of outgoing transitions. The transitions are ordered by selection priority in case more than one transition is simultaneously enabled. A state with no outgoing actions is considered to be a final state.
- EntryActions – A sequence of actions to be performed upon entry to the state.
- ExitActions – A sequence of actions to be performed upon exiting the state.
- StayActions – A sequence of actions to be performed while remaining in the state.
- Memory – An integer memory depth. The depth indicates how many previous states are remembered when the sub-machine encapsulated within the current state is exited. A depth of 0 means that the state sub-machine is memoryless. A depth of 1 means the sub-machine has a shallow history. Depths greater than one are used to bound the depth of the deep history stack (to avoid unbounded state issues).

Junctions

Junctions are used to construct transition graphs. Their only attributes are a set of incoming transition segments and a list of outgoing transition segments (see the definition of transitions below for an explanation of transition segment).

Transitions

Transitions are directed graphs which control changes in state. The attributes of a transition are:

- Sources – The set of states which are a source for any transition segment in the transition.
- Destinations – The set of states which are a destination of transition segment in the transition.
- Segments -- Segments are the edges of the graph; the attributes of a segment are:
    - Source – The start of the segment. Must be a state or junction.
    - Destination -- The end of the segment. Must be a state or junction.
    - Guard – A Boolean condition which must hold true in order for the transition segment to be enabled.
    - ConditionActions – A sequence of actions taken whenever the condition is evaluated to true, regardless of whether or not the transition containing the transition segment actually fires.
    - Actions – A sequence of actions taken whenever the transition containing the transition segment fires.

Actions

An action represents computations which can have side-effects. Actions are further divided into sub-categories:

- Assignment – an assignment action represents the replacement of a stored value. Assignments have two attributes:
    - LExpr – An l-expression which computes a reference to the storage unit to update.
    - RExpr – An expression which computes the value to be stored.
- Signal – A signal is a synchronous message which is multicast to some set of states. Attributes of signals include:
    - Id – As mentioned earlier, all nodes have unique identifier attributes. The id is listed here because it is used to find a receiver for the signal.
    - Destination – a list of states. The signal is sent to the listed states as well as the transitive closure of all their sub-states, so sending a signal to the root state will broadcast the signal to all states in the model.
    - Parameters – a list of expressions, which can be evaluated to compute the actual parameter values.
- Call – A Call is a synchronous message sent to a function or object instead of a state. A return value may be passed back to the sender. The attributes of a call include:
    - Target – A variable (which must hold a reference to an object) or function.
    - Parameters – A list of expressions used to compute the actual parameter values.
    - Return – A variable which will hold the return result.
    - IsTail – If true, execution jumps to the target. The destination of the transition labeled by the call action must be a final state, which should be labeled with the tail call action only (i.e., no other actions are allowed). There should also be no return variable.
- Return – A return action terminates execution of a state machine invoked via a call. It is subject to the same restrictions as tail calls. The only attribute is an expression which is evaluated to compute the value to be returned.

Guards

A guard is used to enable the crossing of transition segments. Guard attributes include:

- EventSet – A list of signals, any one of which will satisfy the event component of the guard. If the list is empty, then the event condition is vacuously true.
- Condition – An expression which yields a Boolean result when evaluated.

Both the event set and the condition must be simultaneously satisfied for the guard to be satisfied.

Variables

Variables are containers for values. The primary attributes of a variable are:

- Type – the type of the variable.
- Initializer -- an expression used to initialize the variable.
- Storage – the domain in which the variable is to be allocated (Static Memory or Stack).

Objects

A variable whose type is a class is a reference to an object.

Functions

Functions provide interfaces which can accept call messages. The actual computation is done by a state machine encapsulated within the function. Attributes of functions include:

- Type – the type of the function.
- Class – if the function is a method, the class to which method belongs.
- This – A variable which contains a hidden parameter used to implement some methods. The type of the variable must be the same as the Class attribute.
- Parameters – a list of variables where incoming actual parameter values will be stored.
- State – the state machine which performs the function computation.

Types

A type is used to infer the set of possible values variable, expression, etc. can hold and what operations can be legally performed on values. The attributes of a type vary depending on a sub-tag, which further refines the type. Type sub-tags and their associated attributes are as follows:

- Atomic -- Atomic types are primarily numeric primitive types with pre-defined semantics. These include integer, floating-point, fixed point, and string. Attributes of atomic types are:
    - Signedness – (integers only) indicates if the type domain includes negative values.
    - Size – Gives the width of the type (e.g. 32 bits, 16 bits, 8 bits, etc.).
    - Bounds – Optional upper and lower bounds on the value domain.
    - Precision – (fixed point) tells the number of digits in the fractional component of a fixed-point value, or (floating point) gives the size of the mantissa.
    - Exponent Range -- (floating point) gives the range of possible exponent values.
- Array – Array types are sequentially allocated storage units, all of the same type. Array attributes are:
    - Element Type – the type of value stored in the array.
    - Bounds – A list of pairs specifying the range of valid index values for each dimension of the array.
    - Organization – Row major or column major.
- Record – Record types are sequentially allocated storage units. Unlike arrays, the storage unit types are heterogeneous. The only attribute of a record is an ordered sequence of element types. (This is because the field names will have already been replaced with field indices at the time the RHL code is generated.)
- Object -- Object types are encapsulated records whose type can be determined at runtime and which can only be accessed via methods defined in the class. The only attribute of an object is its Class.

- Tuple – Tuple types are essentially records without field names. Tuples are primarily used to construct functional types. The only attribute of a tuple is an ordered list of its element types, which must be of length 2 or longer.
- Function – A function type is used describe the input and output domains of a function or method. The attributes of a function are:
    - InputType – The input type of the function.
    - OutputType – The output type of the function
    - SubTag – Indicates whether the type is a function, an object method, or a class method.
    - Class – (methods only) gives the class to which the function belongs.

Classes

A class defines an abstract data type which encapsulates data that only be accessed via a set of supplied methods. Class attributes include:

- Superclass – The parent class of the class. The pre-defined class Object is the root superclass. The class will inherit data members and methods from its parent class.
- Subclasses – A list of the classes of which this class is a parent.
- Methods – a list of functions (all of which must be methods of the class).
- SharedDataMembers – a list of variables, which represent data members shared between all objects of the class.
- ObjectDataMembers – an ordered list of types. This list defines the fields of the record encapsulated within each object of the class.

Expressions

The set of expressions provided by RHL will incorporate standard Boolean and numeric operators, field access for records, array indexing, type casting, constant values for all the primitive types, and variables (i.e., an expression which loads the value of a variable). Operator precedence is not an issue since the expressions will already be in tree-form. Expressions are completely typed. Operators have type signatures, with distinct operators for each type (e.g., there is separate 32-bit integer addition and 64-bit real addition). The sub-expressions of any operators must match the expected input types of the operator (e.g., there is no implicit cast which allows the addition of an integer to a floating-point value, as in C). The interface which supports RHL code generation can be designed to insert these casts automatically to reduce the burden on tools which generate RHL models.

Not included in the set of expression operators are operators with side-effects, such as an increment operator.

L-expressions compute assignable storage locations. The set of RHL l-expressions is small due to the lack of pointers.

Type Checking

The type system for RHL is fairly simple, due to the fully typed expressions, and the lack of a pointer type. The type rules can be briefly summarized as follows:

- A value of any numeric type can be cast to any other numeric type.

- A reference to an object can be cast to a reference to a superclass of the object.
- The type of constant, variable, or function expression is contained in an attribute of the node which defines the constant/variable/function.
- The type of a compound expression is the output type of the operator.
- Numeric and Boolean operators have the same input and output types.
- The type of a cast operation is manifest in the cast.
- For every type T, there are Binary relational operators which operate on two values of type T and produce a Boolean result.

Semantics

The semantics of RHL will be primarily taken from Simulink/Stateflow, as the UML semantics are much sparser (and ambiguous) in comparison. The behavior of state machines will largely follow the behavior of a Stateflow machine with the same structure. The primary difference is the addition of calls to methods. A call to a method is dispatched by consulting the actual (runtime) type of the target object. For example, assume that class *Z* has subclasses *X* and *Y*, all of which implement a method *f()*, and there is a variable *z* which is a reference to an object of class *Z*. When the method *z.f()* is called, the implementation of *f()* which is called could be in any of the classes. If *z* was assigned the result of casting a reference to an object to type *X* to an object of type *Z*, then the *f()* that will be called is the *f()* defined in *X* (not *Z*).

Compilation

A primary requirement of RHL is that it must be possible to compile a model expressed in RHL into a simpler form which can be utilized by verification tools such as Salsa or Reactis. This can be done by using a series of passes which iteratively reduce the complexity of an RHL program. The complexity levels are as follows:

- RHL0 – Full RHL.
- RHL1 – RHL without dynamic method dispatching.
- RHL2 – RHL1 without call messages.
- RHL3 – RHL2 without events.

RHL1 is produced by creating specialized clones of functions for each actual type of object that is passed as a parameter, so that the actual object type can be statically determined. For example, assume that classes *X* and *Y* are subclasses of class *Z*, and that function *f(Z z)* invokes the method *z.g()*. If *f* is called with objects of both type *X* and *Y*, then *f* will be replaced with two variants: *f_X(X z)* and *f_Y(Y z)*. These variants will statically know which instance of the *g* method to call in the *X* class tree (i.e., *f_X* will call *X.g* and *f_Y* will call *Y.g*). This of course means that the callers of *f* will also have to be modified to call one of the specializations of *f*.

RHL2 is produced by inlining all function calls. This is not technically difficult, as there is no recursion and no dynamic message dispatching to worry about. For example, if *f_X(X z)* is called from two call sites with the objects *x1* and *x2* respectively, then *f_X(X z)* will be inlined twice, once at the first call site with *z* replaced by *x1*, and again at the second call site, with *z* replaced by *x2*.

RHL3 is produced by replacing events with variables and expanding the number of transitions. A unique variable is allocated for every event action. The value of this variable is an integer enumeration which designates the transition that will fire when the action is taken. The event actions are then replaced with transitions that set the value of the event variable, as follows: a transition from state q_1 to q_2 labeled with an event action e (and corresponding event variable v_e) is replaced with a two-step transition. The first step is an unguarded transition segment from q_1 into a (new) junction j_1. The second step is taken from a group of parallel transition segments, all of which go to q_2. For every possible receiving transition t_i of the event with guard g_i, a transition segment from j_1 to q_2 with guard g_i is created, whose action is $v_e := i$. One final wrinkle is that the case where no receivers are enabled must be handled. This is done by adding a transition segment from j_1 back to q_1 which has no guard conditions and is prioritized last among all the outgoing transitions segments of j_1.

Once the event senders have been transformed, every event guard is replaced with a test on the corresponding event variable. For example, a transition t_k with a guard waiting for event w (whose event variable is v_w) and condition $x > w$ will have the guard modified to be $(v_w = k)$ *and* $(x > w)$. In order to avoid re-triggering, an action [v_w := -1] must be added to the set of actions performed by transition t_k. As a final note, it is worth pointing out that it is assumed all event variables are initialized to -1, which represents the case where no receiving transition is enabled.

Related Work

In recent years, model-based development has gained credibility in industry as a practical approach to improve software quality. In particular, the industry-defined languages UML and Stateflow/Simulink have grown in popularity as a basis for expressing models of software system architecture and design. The rapid spread of these modeling languages has fueled an interest within the research community to find techniques which can bring the benefits of formal verification methods to models developed in UML and Stateflow/Simulink.

Work related to our efforts includes efforts to formalize the semantics of UML and Stateflow/Simulink. Some attempts have been made to formalize the semantics of Stateflow, such as [Ham05, HR04]. UML has attracted much more attention in this area, due to its many ambiguities. Recent works include [AA00, CF05, FSKdR05, HH05, LS06, TA06]. More relevant to RHL are translation techniques. [SSC$^+$04, JH05] use translation methods as a basis for verification of Simulink model properties.
Certain RSI customers have developed an interest in MWI as a mechanism for interoperation between the Simulink / Stateflow tool set and model-checking tools. In 2005, a team at the University of Minnesota and Rockwell Collins implemented a translator for converting a subset of MWI into an input format for the popular SMV model checker [BCM+92]. The literature on UML translation into verifiable forms is quite extensive and includes [AHK$^+$04, Bos99, CC04, CE06, CF05, DMB02, Esh06, HM05, GH04, KAM06, LCA04, LMM99, MCG$^+$04, PMP01, SB06, SCH02, VGP01].

References

[AÁ00] José Luis Fernández Alemán and José Ambrosio Toval Álvarez. Can intuition become rigorous? foundations for UML model verification tools. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, pages 344–355, San Jose, CA, USA, October 2000. IEEE Computer Society.

[ADB02] I. Majzik A. Darvas and B. Benyó. Verification of uml statechart models of embedded systems. In Strube and et.al., editors, *Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS 2002)*, pages 70–77, Brno, Czech Republic, April 2002.

[AHK$^+$04] Tamarah Arons, Jozef Hooman, Hillel Kugler, Amir Pnueli, and Mark van der Zwaag. Deductive verification of UML models in TLPVS. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference*, volume 3273 of *LNCS*, pages 335–349, Lisbon, Portugal, October 2004. Springer.

[ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electr. Notes Theor. Comput. Sci*, 109:43–56, 2004.

[Bha02] R. Bharadwaj. Sol: A verifiable synchronous language for reactive systems. In Alain Girault Florence Maraninchi and Eric Rutten, editors, *Proc. Synchronous Languages, Applications, and Programming (SLAP '02)*, volume 65 of *Electronic Notes in Theoretical Compute Science*, pages 909–923, Grenoble, France, April 2002. Elsevier.

[BL03] S. Balacco and C. Lanfear. *Embedded Developers' Demand and Requirements for Commercial OSs and Software Development Tools Volume I: Current Practices and Emerging Requirements in the Automotive Vertical Market*. Venture Development Corporation, October 2003.

[Bos99] Prasanta Bose. Automated translation of uml models of architectures for verification and simulation using spin. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 102, Washington, DC, USA, 1999. IEEE Computer Society.

[BS00] R. Bharadwaj and S. Sims. Combining constraint solvers with bdds for automatic invariant checking. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, volume 1785 of *Lecture Notes in Computer Science*, pages 378–394, Berlin, April 2000. Springer-Verlag.

[CC04] J. Chen and H. Cui. Translation from adapted UML to promela for CORBA-based applications. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop*, volume 2989 of

Lecture Notes in Computer Science, pages 234–251, Barcelona, Spain, April 2004. Springer.

[CE06] Manuel Clavel and Marina Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference (AMAST 2006)*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373, Kuressaare, Estonia, July 2006. Springer.

[CF05] M. Couzinier and Louis Féraud. Formal verification of dynamic UML diagrams using TLA+. In Yuri I. Shokin and O. I. Potaturkin, editors, *Proceedings of the Second IASTED International Multi-Conference on Automation, Control, and Information Technology*, pages 85–91, Novosibirsk, Russia, June 2005. IASTED/ACTA Press.

[DLM99] I. Majzik D. Latella and M. Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal Aspects of Computing*, 11(6), 1999.

[Esh06] Rik Eshuis. Symbolic model checking of uml activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65, Manchester, UK, Nov 2005. Springer.

[GH04] Günter Graw and Peter Herrmann. Transformation and verification of executable UML models. *Electr. Notes Theor. Comput. Sci*, 101:3–24, 2004.

[Ham05] Gregoire Hamon. A denotational semantics for stateflow. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172, New York, NY, USA, 2005. ACM Press.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, New York, NY, USA, 1998.

[HR04] G. Hamon and J. Rushby. An operational semantics for stateflow. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering: 7th International Conference (FASE 2004)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, Barcelona, Spain, 2004. Springer-Verlag.

[HS05] Zhaoxia Hu and Sol M. Shatz. A transformation approach for modeling and analysis of complex UML statecharts: A case study. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP-2005)*, volume 1, pages 361–367, Las Vegas, Nevada, USA, June 2005. CSREA Press.

[JH05] Anjali Joshi and Mats Per Erik Heimdahl. Model-based safety analysis of simulink models using scade design verifier. In Rune Winther, Björn Axel

Gran, and Gustav Dahll, editors, *Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2005)*, volume 3688 of *Lecture Notes in Computer Science*, Fredrikstad, Norway, September 2005. Springer.

[KAM06] Ali Kamandi, Mohammad Abdollahi Azgomi, and Ali Movaghar. Transformation of UML models into analyzable OSAN models. *Electr. Notes Theor. Comput. Sci*, 159:3–22, 2006.

[LB03] C. Lanfear and S. Balacco. *The Embedded Software Strategic Market Intelligence Program 2002/2003 – Volume II*. The Venture Development Corporation, March 2003.

[LCA04] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal verification of object-oriented models. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 187–206. Springer, 2004.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[Mat06a] The MathWorks. *Simulink: Simulation and Model-Based Design: User's Guide*. The MathWorks, Sep 2006.

[Mat06b] The MathWorks. *Stateflow and Stateflow Coder: For Use with Simulink: User's Guide*. The MathWorks, Sep 2006.

[MCG+04] Edjard Mota, Edmund M. Clarke, Alex Groce, Waleska Oliveira, Marcia Falcão, and Jorge Kanda. Veriagent: an approach to integrating UML and formal verification tools. *Electr. Notes Theor. Comput. Sci*, 95:111–129, 2004.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual (2nd Edition)*. Addison-Wesley Professional, July 2004.

[SB06] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, January 2006.

[SSC+04] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In Giorgio C. Buttazzo, editor, *Proceedings of the Fourth ACM International Conference On Embedded Software (EMSOFT 2004)*, pages 259–268, Pisa, Italy, September 2004. ACM.

[TA06] Ali Taleghani and Joanne M. Atlee. Semantic variations among UML statemachines. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, (MoDELS-2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 245–259, Genova, Italy, October 2006. Springer.

[VGP01] Dániel Varró, Szilvia Gyapay, and András Pataricza. Automatic transformation of UML models for system verification. In Jon Whittle et al., editors, *WTUML'01: Workshop on Transformations in UML*, pages 123–127, Genova, Italy, April 7th 2001.

[WSH02] Kevin Compton Wuwei Shen and James K. Huggins. A toolset for supporting uml static and dynamic model checking. In *26th International Computer Software and Applications Conference (COMPSAC 2002)*, pages 147–152, Oxford, England, August 2002. IEEE Computer Society.

[ZPP01] István Majzik Zsigmond Pap and András Pataricza. Checking general safety criteria on uml statecharts. In *20th International Conference on Computer Safety, Reliability and Security 2001 (SafeComp 2001)*, number 2187 in Springer Lecture Notes in Computer Science, September 2001.