

Integrating Formal Model Checking with the RTEdge™ AADL Microkernel

2011-01-2531

Published
10/18/2011Serban Gheorghe
Edgewater Computer Systems Inc.

Copyright © 2011 SAE International

doi:10.4271/2011-01-2531

ABSTRACT

Edgewater Computer Systems Inc. product RTEdge™ Platform 1.2 is a software toolset supporting proof based engineering, implementation and deployment of software components, built using the RTEdge™ AADL Microkernel modeling subset. This is a small subset of the AADL component model and execution semantics, covering threads and thread-groups communicating solely through asynchronous event ports and through explicitly shared data ports. Threads behavior is expressed as state machines and dispatch run time semantics is encoded in a Run-time Executive, enforcing pre-emptive priority dispatch based on statically assigned event priorities, with ceiling priority protocol access to shared data. This simple AADL microkernel semantic core can support all dispatch policies, communication and synchronization mechanisms of a fully fledged AADL run time environment, permitting the systematic use of the RTEdge™ static analysis tools for AADL compliant software components. The objective of our work¹ was to integrate RTEdge™ Platform 1.2 with the Promela / Spin (www.spinroot.com) model checker, for the purpose enabling formal verification of user defined formal properties, expressed on software application models built and executed on the RTEdge™ AADL Microkernel. In order to deal with the state space explosion problem, Model Checking relies on abstractions. Human intervention is required in an error prone process of defining the right abstractions, implementing an equivalent abstracted model in the model checker formalism, proving or disproving a property by model checking and applying the findings to the implementation. Assume-Guarantee Compositional Verification provides both a state space “divide” and “conquer” strategy and a model checking automation

strategy, in a reverse engineering process applied to existing component implementations, as an iterative process of assumption refinement by model checking. The RTEdge™ concepts of Protocols, Capsule Interfaces and Composite Capsules open the possibility of using Assume-Guarantee contracts as a specification mechanism, in a forward engineering manner, and of verifying their consistence and conformance by the systematic use of Model Checking. Such an approach would allow verifying very large state spaces by using hierarchical component abstractions formally maintained in the RTEdge™ AADL component model via the “extends” and “implements” refinement relationships. To that end, Protocols, Capsule Interfaces and Composite Capsules must have a compact and convenient way of specifying sets of legal sequences of Signal Events. We developed a set of requirements for adapting IEEE1850 Property Specification Language (PSL) to formally express RTEdge™ software component temporal properties. The adapted PSL will use RTEdge™ Protocols Signals and Data as the underlying vocabulary of the PSL Boolean layer.

INTRODUCTION

The RTEdge™ AADL Microkernel execution semantics is a small and constrained subset of the AADL run-time semantics, well known to the static analysis and code generator tools already built into the RTEdge™ toolset. As such, a companion model checker can be used for either proving formally generic, out-of-the box concrete application properties or for proving formally user-defined, application specific properties.

Typical user workflows for proving a concrete application property via model checking involve creating an abstract model of the application in the formalism of the model

¹This work was funded by the European Space Agency under the 2010 General Studies Partnership Program (GSTP)

checker, specifying the formal property to be proven as assertions or temporal formulas expressed on the abstracted model, and using the model checking engine to search for a counter-example proving that the property is invalid.

The user workflow for proving formally an application property would be greatly simplified if such formal properties could be expressed as annotations added and understood in the context of the concrete application. They can further be simplified if counter-example findings on the abstract model are interpreted and played back as an execution trace in the context of the concrete application.

Although the user must understand various translation abstraction options, the error prone process and inherent complexity of obtaining a viable abstraction model and interpreting findings could be automated and by in large hidden from the end user.

Our work is predicated on the assumption that there are two essential key factors to speed up the adoption of formal model checking and bring it into the mainstream software engineering practice:

1. Providing adequate tool support for choosing and generating the right abstracted model for a concrete application
2. Automating some of the error prone user workflows involved in translating and interpreting results between the abstracted model and the concrete application.

Figure 1 shows the user workflow for verifying a formal property on an RTEdge™ Application using the Promela/Spin model checker, as well as the language and tool automation support provided by our project. As shown in Figure 1, there were four relevant technical steps we pursued during this project,:

1. Define RTEdge™ formal annotations; these are RTEdge™ language extensions for expressing assertions and temporal properties on a concrete RTEdge™ Application
2. Provide automatic code generation from an RTEdge™ concrete Application to an abstracted Promela Model, using different abstraction options settings
3. Define simple mappings of formal properties expressed as annotations on the RTEdge™ Application into the Promela modeling domain
4. Map back Spin assertion violations notifications and counter-example files into the RTEdge™ Application state machines, by using meta-data embedded in the Promela generated code

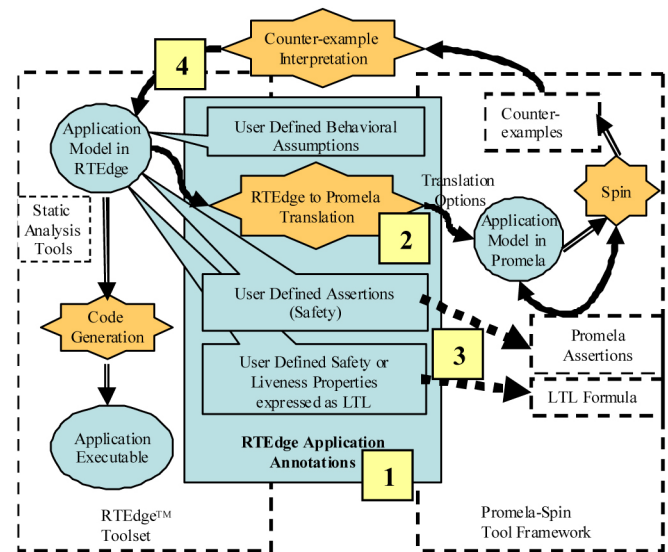


Figure 1. Formal Model Checking User Workflow in RTEdge™

Spin is a mature model checker and its companion language Promela assumes execution semantics very close to the RTEdge™ AADL Microkernel. The Spin model checker has also another two attractive features when used as a companion model checker for RTEdge™ software applications:

1. Our approach capitalizes on the fact that both RTEdge and Spin use C as an underlying programming language. This greatly simplifies data type mapping between the concrete and the abstract application model, by allowing RTEdge™ elementary types to be mapped directly into Promela elementary types and resolved at execution time as C type variables.
2. We also use the unique Spin feature of embedding in the Verifier the same C code running in the concrete implementation, by using the built-in Spin functions `c_code`, `c_track`, etc... This approach eliminates a major source of errors in the Model Checking process and applies automation to obtaining precise state transition functions in the abstract model. It also provides an opportunity for fine granularity behavior abstraction, at the level of state machine actions expressed as C functions, while mapping exactly the RTEdge™ state machines from the concrete application into the abstracted model.

RTEdge™ TOOLSET BACKGROUND

RTEdge™ is a domain specific toolset for critical real-time software applications, targeting distributed software applications with discrete, time bounded inputs and outputs. It is a proof based, Model Driven Development environment, incorporating static model analysis for real-time scheduling

and inter-working with model checker tools for proving functional properties.

In all engineering disciplines except Software Engineering, by necessity model abstractions are distinct and are built upon a different engineering medium than the system they abstract. Software Engineering is the only discipline which allows refining models into full implementations without changing the engineering medium, tools or methods [BS6]; in other words, the model serves as a “single source of truth” for tools proving static properties and for tools generating an executable implementation. RTEdge™ AADL Microkernel toolset is built on this premise, based on a small modeling language subset of AADL chosen such that it meets the following objectives:

1. It permits capturing precise specification and refinement information:

- software components interface specifications
- specifications of software component architectures, either as a composition or decomposition of software components
- software components refinement into full fledged implementations on specific hardware targets
- software components integration into executable applications deployable on target hardware platforms

The RTEdge™ AADL Microkernel is using a subset of the *AADL software component model*: threads and thread groups communicating through Ports, with separate inheritance trees for component types and component implementations. The inheritance trees are based on the “extends” refinement relationship and linked to each other via “implements” relationships.

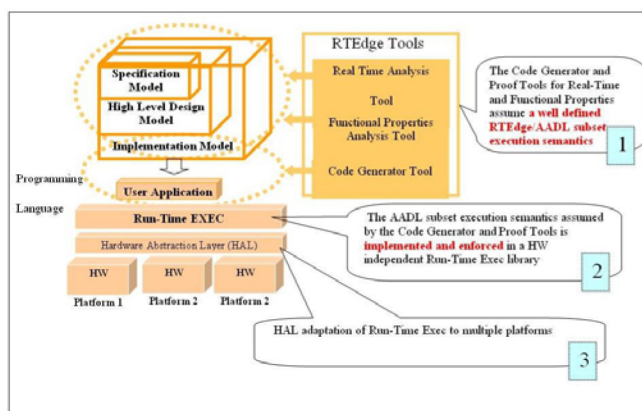


Figure 2. Model Refinement, Application Code Generation and Run-Time execution semantic enforcement

By using model refinement, an Implementation Model will be obtained from an included set of types of a High Level Design Model, which in turn can be obtained from a subset of

root types of a Specification Model. This approach ensures that higher level models are kept always in sync with lower level detailed models, as lower level component types can be traced back to higher level component types, as shown in [Figure 2](#).

2. It provides clear execution semantics:

- Operational semantics is encoded into a hardware independent run-time executive library using a Hardware Abstraction Layer that can be ported to any preemptive priority RTOS platform
- The execution semantics is well known to static analysis tools proving functional properties and real-time constraints. This approach allows the design of very powerful static analysis tools; it protects the investment and increases the trust in static analysis tools, by permitting their consistent use across the development life-cycle.

3. It provides adequate life cycle development support for

- developing code in the context of the model
- automatic Application code generation and load build
- debugging at the model level
- execution instrumentation and monitoring functions

This type of development support ensures that executable code is *always produced, modified and understood in the context of the model*.

STATIC PROOF BASED TOOLS

RTEdge™ toolset uses the AADL concept of Event Flow as the base for static resource allocation. User specifies execution costs and Application input/output Flow deadlines. Static real-time analysis tools are provided to prove that competing sets of Event Flows are schedulable. The RTEdge™ Scheduling Tool calculates Worst Case Response Time (WCRT) for each Flow originated by an input event, using a modified version of the Harbour-Klein-Lehoczky [HKL14] scheduling algorithm of task sets with precedence.

The following proof based engineering and analysis tools are available in RTEdge™ :

1. Hard real-time software Applications timing constraints are specified using AADL Event Flows and proven at the *Flow granularity level*. This is achieved by using *priority based mathematical scheduling models* for processor and communication channels embedded in the tool.
2. Buffer pool sizes for communicating Ports are proven to be bounded based on defined event arrival rates, through global Event Flow analysis.

3. Engineering of the capacity of soft real-time Event Flows and soft-real time servers is supported at design time and enforced at run-time.
4. Support is provided for the formal specification of *component interface contracts (Protocols and Capsule Interfaces)* and for proving implementation conformance through static analysis. The inter-working with the Spin model checker enhances this capability, enabling the expression of RTEdge™ Application properties as Linear Temporal Logic formulas and proving them by formal model checking.
5. Executable code generation to the RTEdge™ Runtime Executive is provided, enforcing a very precise execution semantics subset of AADL, guaranteed to match the assumptions of the static analysis tools.

The Flow WCRT Calculator of the RTEdge™ Scheduling Tool makes the assumption that the user Application model is “well behaved”, i.e. the arrival of a new input in any state of the Application will not create infinite loops (live-lock) or result in execution blockage of the cooperating concurrent state machines (deadlock). We call this the Transaction Completion property. One of the intended and achieved benefits of our work was to express this property as a Linear Temporal Logic (LTL) formula and automate the proof of this property via Spin model checking. We also allow the RTEdge™ Application designer to express and prove other custom application specific LTL properties and assertions.

RTEdge™ EXECUTION SEMANTICS

RTEdge™ uses a microkernel approach to both modeling and run-time execution semantics, defining a small AADL modeling and execution semantics subset such that

- a. The AADL refinement based component model is supported for software components.
- b. All the AADL thread dispatch policies, communication and synchronization patterns and dispatch enforcements can be emulated based on the semantics of the RTEdge™ AADL microkernel.

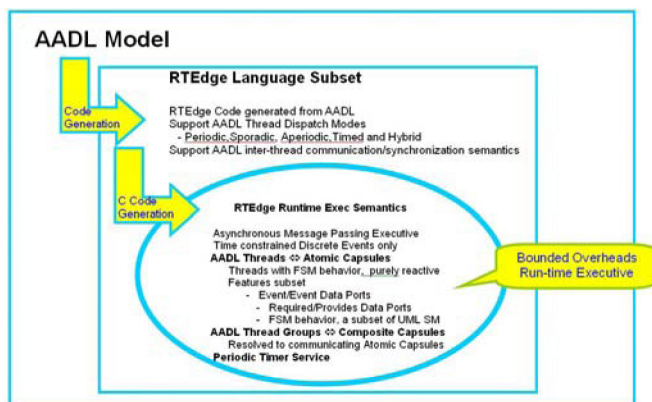


Figure 3. RTEdge™ as an AADL Microkernel

The run-time semantics of the RTEdge™ AADL Microkernel modeling subset is based on the simple observation that any inter-thread communications and synchronization mechanisms can be expressed as pre-defined sequences of asynchronous messages (called *Protocol contracts*) exchanged between threads with state machine behavior and explicit access to shared data.

The RTEdge AADL Microkernel run-time semantics is very similar to well-known profile subsets like ADA Ravenscar, using a preemptive priority based thread dispatch policy, with static allocated priorities and static enforcement of the priority ceiling protocol. The inter-thread communications is however based on connected AADL Event/Event Data/Data port groups.

Priority assignment is statically performed at a finer grain than in Ravenscar, at the event granularity level. It is based on user specified Event Flow arrival rates and deadlines, using Deadline Monotonic Scheduling and adjusted statically for Ceiling Priority based on shared access to Data Ports. As a result, the same thread can operate at multiple statically allocated priorities, depending on the priority of the event it reacts to and the action it executes in a particular state.

Arrival time constraint specifications are attached to events, not to threads. Periodic events are dispatched to interested threads the same way as events with aperiodic or bursty sporadic time arrival specification. Periodic thread behavior results from reacting to periodic events. An RTEdge™ Atomic Capsule (AC) is equivalent to an AADL aperiodic Thread with Event and Data Port group features and with behavior described by an extended Finite State Machine (FSM). An RTEdge™ Composite Capsule is equivalent to an AADL Thread Group, a vehicle for interconnecting and composing hierarchically other Atomic or Composite Capsules.

The RTEdge™/AADL Runtime is easily portable to multiple hardware platforms and has bounded overheads that can be measured on any hardware platform and used in the platform specific scheduling mathematical models.

ATOMIC CAPSULES FINITE STATE MACHINES

At execution time, an RTEdge™ Application is reduced to a set of concurrent AC instances communicating through Ports connected by point-to-point connections i.e. a collection of concurrent extended FSM communicating through pairs of point-to-point unidirectional First-In-First-Out (FIFO) channels. Each AC Port P_i corresponds to a pair of FIFO channels queuing up only input Signal Events Sig_{Pr_i} declared in the associated Protocol base role Pr_i or input signal events $Sig_{\bar{Pr}_i}$ of its conjugated complement role $\bar{Pr}_i$. Each signal in

Sig_{pr_i} and Sig_{pr_i} has an associated message type that can be either an RTEdge™ built-in type or void.

Assuming without loss of generality that all Ports of an AC are typed by a Protocol base role, the input Port Signal alphabet accepted by the AC extended FSM is

$$I = \bigcup_i P_i.Sig_{pr_i} \text{ with } 1 \leq i \leq N,$$

where N is the total number of AC ports, and the output Port Signal alphabet is

$$O = \bigcup_i P_i.Sig_{pr_i} \text{ with } 1 \leq i \leq N.$$

Definition 1. An AC extended FSM M_i is a t -uple $\{I, O, LV, SD, SS, TS, UP, EC, ST, TT, IS, ES\}$ where I and O are the Input and Output Port Signal sets as defined above.

LV is a n -dimensional Cartesian product of the Local Variables type domains, with each Local Variable typed by a RTEdge™ built-in elementary type:

$LV = LV_1 LV_2 \dots LV_n$ where LV_i is the type of the i -th AC local variable

SD is a k -dimensional Cartesian Product of “required” Shared Data Access Points (DAP) type domains

$SD = SD_1 SD_2 \dots SD_k$ where SD_i is the type of the i -th “required” DAP

SS is the set of Stable States

TS is the set of Transient States

UP is a set of Local Variables Update Functions $UP : LV \times SD \times LastMsg \rightarrow LV \times SD$, where $LastMsg \in I$ is the type of the last received input signal event message data

EC is a set of Execution Conditions $EC : LV \times SD \rightarrow \{0,1\}$

ST is the transition relation from Stable States $ST : SS \times I^w \rightarrow \{SS \cup TS\} \times O^m$

where w is the cardinality of the set I , i.e. the number of possible different incoming Port Signals
 m is a natural number bound on output signal events sent during a transition

TT is the transition relation from Transient States $TT : TS \times UP \times EC^p \rightarrow \{SS \cup TS\} \times O^m$

where p is the cardinality of the set of Stable and Transient States $SS \cup TS$
 m is the natural number bound of output signal events sent during a transition

$IS \in SS$ is the initial state, a distinguished Stable State

$ES \in SS$ is an exception state, a distinguished Stable State

To ensure run-time determinism, both ST and TT transition relation are constrained to become function by imposing extra condition in the RTEdge™ run time execution semantics.

ST becomes a function by ensuring that a unique Port Signal trigger label is always selected for transitions out-flowing from a Stable State, when multiple signal events are present in the incoming Port FIFOs:

- Only the incoming Signal Events at the head of the FIFO queues associated with Ports are considered as Port Signal triggers and are consumed on dispatch. Other signal events queued on the incoming FIFO will be considered in future Stable States.
- When multiple Signal Events are present at the head of incoming FIFOs from different Ports, and each of them appears as a Port Signal label of transitions out-flowing from a Stable State, the Port Signal trigger with the highest Priority is selected. For equal Priority, a tie-break rule is used to select the Port Signal trigger. The Run-Time Executive is using the oldest Port Signal trigger event (i.e. with the earliest timestamp) as a tie-break rule.
- If a Port is deferred in a Stable State, all incoming Signals from that port are not considered as a dispatch trigger from that Stable State
- If a Port Signal is present in a Stable State for which it is neither a trigger label, nor declared as a deferred Port, a transition to a default error state ES “undefined trigger exception” is performed.

TT becomes a function by ensuring always that, out of all the Execution Condition functions labeling the transitions out-flowing from a Transient State, at most one takes the value 1, while all the other Execution Conditions take the value 0 (deterministic outcome). A *Default* attribute on one of the labels ensures that there is always a transition enabled out of a Transient State.

Atomic Capsule Activities And Completion Events

Assuming that a Transient State has n possible outgoing transitions, it means that they are n non-trivial expression condition functions $ec_k \in EC$ $ec_k : LV \times SD \rightarrow \{0,1\}$, $1 \leq k \leq n$ that can possible take the value 1, all the other impossible

transitions to other states getting an implicit constant 0 expression condition label.

In the RTEdge™ toolset, the update functions UP and the Execution Conditions functions in EC are implemented as one *Atomic Capsule Activity*, coded as a C language function, associated with a Transient State and returning an enumerated return code $rc \in \{val_1, val_2, \dots, val_n\}$ such that one and only one of the $rc = val_k$ Boolean Execution Condition labels on the transitions out-flowing from the Transient State is True and all the other are False.

The AC Activity C function defines an enumeration set of integers $RC_VALUES = \{val_1, val_2, \dots, val_n\}$, with $val_k \in I$, corresponding to the set of non-trivial Execution Condition functions $ec_k \in EC$, $ec_k : LV \times SD \rightarrow \{0,1\}$, $1 \leq k \leq n$, and implements a return code function

$$rc(ec_1, \dots, ec_n) \rightarrow RC_VALUES \text{ defined as}$$

$$rc(ec_1, \dots, ec_n) = \begin{cases} val_k & \text{iff } ec_k \text{ evaluates to } 1 \\ val_{default} & 1 \leq default \leq n \text{ iff all } ec_k \text{ evaluate } 0 \end{cases}$$

We will call the simpler Boolean expression conditions $rc=val_k$ "Return Code Condition" labels.

Definition 2. We call the evaluation of the Return Code Condition label at the completion of an Activity an *Activity Completion Event*.

Activity Completion Events determine the control flow out of a Transient State.

AC FSM AS A LABEL TRANSITION SYSTEM

Note that Definition 1 is compatible with a classic Finite State Automaton described by a Label Transition System (LTS), i.e. a tuple $\{S, s_0, L, T, F\}$, where

$S = I \times LV \times SD \times SS \times TS$ is the finite set of states

$s_0 = IS$ is the initial state

$L = I \cup O \cup UP \cup EC$ defines the finite set of Labels

$T = ST \cup TT$ is the set of transitions

$F = SS$ is the set of Final States; note that any Stable State can be a Final State in the Finite State Automaton equivalent to an AC extended FSM.

We will use the following notation to show transition labeling:

$$t : ss_1 \xrightarrow{P_1 ? a / P_2 ! bc} ss_2$$

represents a transition from Stable State ss_1 to state ss_2 with consumption of signal event a from incoming FIFO channel of Port P_1 and with sending (appending) to the outgoing FIFO channel connected to Port P_2 two signals b and c in this order. As Event Ports of any ACs are connected one-to-one, it is assumed that the access to the Port FIFO channels is exclusive, i.e., for every FIFO channel at most one AC FSM can send to (or consume from, respectively) the FIFO channel. Independent Application Input signal events arrive in FIFO channels written by outside entities, and Application Output signal events go to FIFO channels that are consumed by outside entities.

$$t : ts_1 \xrightarrow{act.rc=k / P_2 ! bc} s_3$$

represents a transition from a Transient State ts_1 to another state s_3 , labeled by a Return Code Boolean expression of the Activity associated to the Transient State.

RTEdge™ TRANSACTIONS

The arrival of an Application Input signal event on a RTEdge™ Application Port will cause a sequence of transitions in the underlying Atomic Capsule state machines causally related either via signal events sent between Atomic Capsules or related via intra Atomic Capsule Activity Completion Events, eventually resulting in multiple Application Outputs being produced.

Definition 3: By global state G^k of an Application we mean the joint state of

- All Atomic Capsule (AC) instances FSM's state
 - All AC Instances Local Variables and shared data values
 - All signal event FIFOs
- at the same point in time.

Definition 4: By a Transition we mean a FSM transition of an AC instance present in the RTEdge™ Application.

Definition 5: An execution path of the RTEdge™ Application is a possible infinite sequence of Transitions starting in the initial state:

$$START = G^0 \xrightarrow{t_0} G^1 \xrightarrow{t_1} G^2 \xrightarrow{t_2} \dots G^k \xrightarrow{t_k} \dots$$

We call an occurrence of a transition within an execution path a *Transition Instance*, abbreviated as $t^0, t^1, t^2, \dots, t^k \dots$