Combining Formal Refinement and Model Checking for Real-Time Systems Verification

Alexander Krupp, Wolfgang Mueller
Paderborn University, Germany
Alexander.Krupp@c-lab.de, Wolfgang.Mueller@c-lab.de

Abstract

We present a framework, which combines model checking and theorem prover based refinement for real-time systems design focusing on the refinement of non deterministic state oriented to timed state oriented models. Our verification flow starts from a cycle accurate finite state machine for the RAVEN model checker. We present a translation, which transforms the model into efficient B language code. After refining the RAVEN model and annotating it, the time accurate model is also translated to B so that the B theorem prover can verify the refinement almost automatically. The experimental results demonstrate that the presented solution requires almost no user interaction and results in most efficient proof time when applying the Atelier-B theorem prover.

1 Introduction

With the increasing complexity of systems on a chip, formal verification has started to play a significant role in systems design. In hardware design, and partly in software design, equivalence checks and model checking are applied on a regular basis. For software design, the B method has been successfully applied in several industrial projects. Additionally, model checking receives increasing acceptance in embedded software design. However, theorem proving in general is less accepted, which is mainly due to the fact that it still requires too many user interactions conducted by educated experts.

This paper describes an efficient combination of model checking and theorem proven refinement based on the RAVEN model checker [Ruf01] and the Atelier-B toolset [Abr96]. We present a model checking oriented verification flow, which is based on automatic translation to B for refinement verification. Since our investigations concern real-time system verification, we focus our interest on the refinement of a cycle accurate model into a time accurate model [WS96], which can be further refined to an implementation. In that context, we present a B code generation, which results in a very efficient application of the Atelier-B theorem prover. In contrast to related approaches, our experimental results demonstrate that the proof of the generated code requires almost no interaction with the prover and additionally results in very low runtimes.

The remainder of this paper is structured as follows. The next section discusses related work. Section 3 and 4 introduce RAVEN and B before Section 5 presents our approach combining RAVEN and B to a unified framework. Thereafter, we present our experimental results before the final section closes with a conclusion and future work.

2 Related Work

There is significant work integrating model checkers into theorem provers and vice versa.

PVS (Prototype Verification System) from SRI (Stanford Research Institute) is basically a theorem prover where the PVS specification language is based on high order predicate logic. Shankar et al. enhance PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem
proving, and model checking to separate concerns as well as to compute concurrent systems properties [Sha00]. STeP (Stanford Temporal Prover) was developed by Manna et al. at Stanford University implemented in Standard ML and C [Man94]. STeP integrates a model checker into an automatic deductive theorem prover. The input for model checking is given as a set of temporal formulae and a transition system, which is generated from a description in a reactive system specification language (SPL) or a description of a VHDL subset. More recently, Berezin has introduced the SyMP framework, which integrates a model checker into a HOL based theorem prover for general investigations on effectiveness and efficiency [Ber02]. His work is on the applicability for domain specific computer assisted manual proofs where main examples come from hardware design.

Mocha from UC Berkeley [AH96] is basically a model checker enhanced by a theorem prover and a simulator to provide an interactive environment for concurrent system specification and verification. However, in the Mocha framework theorem proving is still highly interactive and no efficient reasoning of more complex systems is reported.

In the context of the B theorem prover, Mikhailov and Butler combine theorem proving and constraint solving [MB02]. They focus on the B theorem prover and the Alloy Constraint Analyser for general property verification. Fokkink et al. employ the B method and combine it with μCRL [BFG+01]. They describe the use of B refinement in combination with model checking to arrive at a formally verified prototype implementation of a data acquisition system of the Lynx military helicopters. They present the refinement of a system implementation from a first abstract property specification.

All approaches consider timeless models and do not cover refinement with respect to real-time properties in finite state machines. Only Zandin has investigated real-time property specification with B by the example of a cruise controller in [Zan99]. However, he reports problems with respect to the complexity of the proof during refinement.

In contrast to the HOL based approaches, we present a model checking based approach with the RAVEN model checker in conjunction with the Atelier-B theorem prover for formal refinement with very little manual interaction. We focus on the verification of real-time systems and on the refinement from cycle accurate to time accurate models based on an efficient mapping from the RAVEN Input Language to the B language.

3 Real-Time Model Checking with RAVEN

Model checking in the domain of electronic design automation is due to the pioneering work of Clarke et al. in [CE81] and their SMV model checker. SMV verifies a given set of synchronously communicating state machines with respect to properties given by a set of formulae in tree temporal logic, namely CTL (Computational Tree Logic).

For our approach, we apply the RAVEN (Real-Time Analyzing and Verification Environment) real-time model checker, which extends basic model checking for real-time systems verification by additional analysis algorithms [Ruf01]. In RAVEN, a model is given by I/O-Interval Structures and the specification by CCTL (Clocked CTL).

I/O-Interval Structures are based on Kripke structures with \([\text{min, max}]\)-time intervals at the state transitions. Executing interval structures proceeds as follows. Assume that each interval structure has exactly one clock for measuring time. The clock is reset to zero, if a new state is entered. A state may be left, if the actual clock value corresponds to a delay time labeled at an outgoing transition. The state must be left where the maximal delay time of all outgoing transitions is reached.

CCTL is a time bounded temporal logic. In contrast to classical CTL, the temporal operators \(F\) (i.e., eventually), \(G\) (globally), and \(U\) (until) are provided with interval time bounds \([a, b], a \in \mathbb{N_0}, b \in \mathbb{N_0} \cup \{\infty}\). The symbol \(\infty\) is defined through: \(\forall i \in \mathbb{N_0} : i < \infty\). These temporal operators can also have a single time bound only. If no interval values are specified, the lower bound is zero and the upper
bound is infinity by default. The X-operator (i.e., next) can have a single time bound \([a]\) only \((a \in \mathbb{N})\). If no time bound is specified, it is implicitly set to one.

In RAVEN, I/O-Interval Structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL specification contains (a) a set of global definitions, e.g., fixed time bounds or frequently used formulae, (b) the specification of parallel running modules, i.e., a textual specification of I/O-Interval Structures, and (c) a set of CCTL formulae, representing required properties of the model. The following code is a fragment of a RIL model with 4 states and two state transition from wait to failed and from wait to accept.

```
MODULE consumer
SIGNAL
  state : (wait, reject, accept, failed)
INPUTS loadFail := GlobalFailure
  loaderIdle := (loader.state=loader.idle)
DEFINE rejectOrder := (state = reject)
INIT state = wait
TRANS
  |- state=wait
  -- loadFail       --> state=failed
  -- !loadFail & loaderIdle --> state=accept
  ...
```

For CCTL property specification, consider the following example, which defines that the input buffer of a consumer must not be blocked in order to guarantee sufficient continuous workload, i.e., each accepted delivery request must be followed by actually loading an item at the input buffer within 100 time units after acceptance:

```
AG((consumer.state = consumer.accept)
  \rightarrow AF[100]((loader.state = loader.wait)
  \& AX(loader.state = loader.load))
)
```

4 Refinement with B

The Boyer–Moore Theorem Prover (BMTP) and HOL are the two classical approaches to theorem proving in the domain of electronic design automation. BMTP and HOL are both interactive proof assistants for higher order logic [BM88, Gor93]. In theorem proving, a proof has to be interactively derived from a set of axioms and inference rules. Though several practical studies have been undertaken, interactive theorem proving has not received wider industrial acceptance so far.

On the other hand, B specification and its theorem proving has been successfully applied in industrial projects. Here, one of the biggest projects was the development of a controller for the Parisian driverless metro Meteor by MATRA in 1992.

B stands for a methodology, a language, and a toolset for the specification, design, and coding of software systems introduced by Abrial in [Abr96].

B is based on viewing a program as a mathematical object and the concepts of pre- and postconditions, of non determinism, and weakest precondition similar to VDM and Z.

The B language is based on the concept of abstract machines (AMs). An abstract machine consists of VARIABLES and OPERATIONS and is defined as follows

```
MACHINE M( ... )
CONSTRANTS
...
```
The variables represent the state of the machine, which is constrained by the INVARIANT. Operations may change the machine’s state and they may return a list of results. An AM is refined by reducing non determinism and abstract functions until a deterministic implementation is reached, which may be translated into executable code.

The B method is based on the notion of refinement of specifications supported by the B language. Refinement means the replacement of a machine \( M \) by a machine \( N \) and that the operations of \( M \) (with identical signatures) are defined by \( N \) (but not necessarily vice versa). Syntactically, a refinement is defined as

\[
\text{REFINEMENT } N \text{ REFINES } M
\]

Operations with identical signatures are given in \( N \), but they are working on a different internal state or a different specification of the operation. The important requirement is that the user must be able to use machine \( N \) as machine \( M \). The proof obligation for operation refinement in \( B \) is

\[
\forall (y, z) \cdot (I_M \land I_N \land P_M \Rightarrow P_N \land ([r := r'] S_N) \Rightarrow [S_M] - (I_N \land r = r'))
\]

with abstract and refined state \( y, z \), invariants of abstract and refined machine \( I_M, I_N \), preconditions of abstract and refined substitution \( P_M, P_N \), abstract and refined substitution \( S_M, S_N \) and a result \( r \) of \( S_M, S_N \). The interpretation of this equation is, that \( S_N \) does never produce a state and result \( r \), which cannot be reached by the original substitution \( S_M \) with respect to the linking invariant \( I_N \).

Refinement can be conducted in three different ways: (i) the removal of the non executable elements of the pseudo code (pre condition and non deterministic choice), (ii) the introduction of the classical control structures of programming (sequencing and loop), and (iii) the transformation of data structures (sets, relations, functions, sequences and trees, ...).

A final refinement defines the implementation of the system given by

\[
\text{IMPLEMENTATION } I \text{ REFINES } M
\]

At implementation level, which is also denoted as B0 (‘B Zero’), the B definition looks very much like a PASCAL program. From B0, the B toolset can automatically generate Ada, C, or C++.

5 Real-Time Systems Verification and Refinement

Today’s systems design typically starts with state oriented models. For several applications, StateCharts are frequently applied as an advanced graphical front-end for documentation, model checking, as well as for VHDL, Verilog and C code generation. In our verification environment, we apply StateCharts as graphical capture and generate RAVEN RIL code. At that level, we can investigate the cycle accurate models by simulation and checking their properties through model checking by the RAVEN model checker. After first verifications, we automatically generate a corresponding B code model in order to prove the correctness of the further refinements. Here, our focus is on the refinement of the RIL model through annotation of transitions with timing specification and minor modification in state transitions.
like removal of self loops. We denote the time annotated RIL model as RIL\textsuperscript{T}. When generating B code from RIL\textsuperscript{T} resulting in B\textsuperscript{T}, the B prover verifies whether the B\textsuperscript{T} model is a refinement of the B model, which also verifies that the RIL\textsuperscript{T} model is a correct refinement of the RIL model. Through the B environment, further refinement to the B implementation level B0 is possible, from which we can automatically generate C code for a correct implementation.

Figure 1 gives an overview of the complete design flow from StateCharts over RIL to time annotated RIL\textsuperscript{T} and, correspondingly, over B and B\textsuperscript{T} to C code generation. The individual steps of this flow are outlined in more detail in the following subsections before the next section gives experimental results by means of a small example.

5.1 RIL Code Generation

In a first step, the StateChart model is mapped to RIL. We do not consider complete StateCharts as they are given in UML. We rather limit our scope to a practical StateCharts subset, which corresponds to hierarchical finite state machines as implemented by Statemate or in the SMV code generation presented in [CH00], e.g., interlevel transitions are not supported. The state hierarchies are basically mapped into RIL modules and their connections by INPUTs and DEFINEs to variables. For state transitions, guards are limited to Boolean and Integer variables where actions define their updates. This limitation is basically due to the applied refinement, which does not efficiently apply for the automatic refinement of more complex data types in first design steps.

5.2 B Generation

Transforming RIL models into B machines requires basic restructuring. For an efficient proof, we break down the generated code into two B specifications, i.e., two for each RIL module. The first one covers structural definitions and the second one adds behavioral definitions. The separation into two B machines makes these machines better manageable by the B prover, which generates much less proof obligations for each machine. The structural B machines cover the data dependencies defined by INPUTs and DEFINEs as well as the declaration of state variables (SIGNA\textsc{ls}) and their range. Invariants additionally define data consistencies of the data, which are propagated between the different machines.

The B machines covering the module’s behavior define the corresponding guarded state transition and their actions, which are mapped to one operation \textit{doTransition}. The following code fragment gives a
simple example of a state machine with a transition from state $s=\text{produce}$ to $\text{ready}$ and sketches a second transition from state $s=\text{ready}$. An additional temporary variable $s_{\text{tmp}}$ in combination with the ANY operator is used to implement non deterministic state transitions.

\begin{verbatim}
MACHINE producer1_B_BEHAVIOR
... 
OPERATIONS
doTransition =
  IF s = produce THEN
  ANY s_{\text{tmp}} ...
  WHERE
  s_{\text{tmp}} : s &
  s_{\text{tmp}} : \{\text{produce,ready}\} & ...
  THEN s := s_{\text{tmp}} || ...
  END
ELSIF s = ready THEN
...
...
END
\end{verbatim}

The individual $doTransition$ operations are triggered by one extra B machine, i.e., EXEC, which implements the computation model of RAVEN's I/O-Interval Structures.\textsuperscript{1} That machine basically implements two computational steps:

1. value propagation (RIL DEFINEs) to corresponding INPUTs of other modules
2. transition execution through $doTransition$

Additional invariants in the EXEC machine guarantee that these steps are executed in a specific order.

### 5.3 RIL Refinement

When generating RIL from StateCharts, we arrive at synchronously communicating, non deterministic finite state machines, which are executed at cycle accurate basis.

For real-time system specification and verification, those state machines are annotated in RIL with timing information and transitions are optimized and specified in details so that we obtain time annotated, deterministic finite state machines and denote their notation as RIL.\textsuperscript{2} Consider the following example of a non deterministic transition from state $s=\text{produce}$ to $\text{ready/produce}$.

\begin{verbatim}
| s=\text{produce} -- | s := \text{ready} 
| s := \text{produce} |
\end{verbatim}

The transition can be refined to a timed transition with time T1 such as

\begin{verbatim}
| s=\text{produce} -- :T1 | s := \text{ready} |
\end{verbatim}

This transition fires after T1 time steps, and changes to state $s=\text{ready}$. It remains in state $\text{produce}$ before the timer for T1 has expired. Because the transition does not lead to a state other than $\text{ready}$ or $\text{produce}$, it implements an actual refinement of the previous specification.

\textsuperscript{1}For technical matters we require yet another B machine, which starts computation but which is ignored in the remainder of this paper.
5.4 $B^T$ Generation

Starting from $RIL^T$, we can automatically generate the corresponding B code, i.e., $B^T$, as a refinement of the previously generated B so that the B prover can verify the refinement step. The $B^T$ code generation performs similar to the B code generation with additional instantiation of a Timer and TIME constants for each machine and the additional code in doTransition for managing (advancing and resetting) the timer.

Consider the example given in subsection 5.2 again and the refinement of the previous subsection by adding a time delay T1 to the transition. The following code defines producer1_BT as a refinement of producer1_B_BEHAVIOR. The refinement instantiates a timer with delay T1. In the first transition, the transition definition is encapsulated and executed only when the timer has expired. Otherwise, the execution is skipped. If expired, the timer is reset, otherwise it is advanced until it reached its upper bound and expires.

```
REFINEMENT producer1_BT
REFINES producer1_B_BEHAVIOR
...
INCLUDES
   TIMER_produceT1.Timer(TIMES_produceT1)
...
OPERATIONS
doTransition =
BEGIN
   IF s = produce THEN
      IF TIMER_produceT1.expired = TRUE THEN
         ANY ... WHERE s_tmp = ready & ...
         THEN s := s_tmp || ... END
      ELSE skip END || ...
      IF TIMER_produceT1.expired = TRUE THEN
         TIMER_produceT1.doReset
      ELSE TIMER_produceT1.doAdvance END
      ELIF s = ready THEN
      ...
   END
```

With the given B behavioral model and its time annotated refinement, the B system now can generate proof obligations, which are verified by the B prover.

5.5 $B^T$ Refinement and C Code Generation

Refining $B^T$ to B0 basically means replacing simultaneous by sequential substitution and replacing non deterministic predicates and statements by deterministic ones. This results in substantial changes of the example operation code. From a programmer’s point of view the operation doTransition description is now very straight forward, as the semantics of the B0 language is very similar to traditional sequential programming languages where the state transitions are given by embedded if-then-else statements.

```
IMPLEMENTATION producer1_B0
REFINES producer1_BT
...
doTransition =
BEGIN
   IF s = produce THEN
      IF TIMER_produceT1.expired = TRUE THEN
         s := ready;
         TIMER_produceT1.doReset
      ELSE
          s := s;
```
TIMER_produceT1.doAdvance
END
ELSIF s = ready THEN
  s := produce
END;
END

At that level, the code is well structured and can be easily managed by a programmer. The proof of the implementation for B0 can be automatically accomplished with Atelier-B. After successful verification, proven C code can be automatically generated.

6 Experimental Results

Several approaches for generating B code from RIL and their refinement have been tested with Atelier-B before arriving at the approach presented in the previous section. Experiments were based on a very simple producer-consumer example with additional buffer and maintenance intervals as given in Fig. 2. The example has 6 concurrent substates with 13 states and 13 transitions in total.

![Figure 2: Time Annotated StateChart](image)

First approaches gave huge numbers of generated proof obligations with sometimes up to 90,000 proof obligations, resulting in a not acceptable runtime of more than 22 hours on a Sun Enterprise 450/4400 (1GB RAM, 400MHz UltraSparc II). The experiments have shown that successful proof of the composed system in B very much depends on the separation of the specification into different B machines so that we finally arrived at a very efficient separation.

The experimental results based on the presented approach seem to be really promising with a runtime of 41 seconds for the generation of proof obligations and additional 48 seconds for the proof. Table 1 shows refinement details of the example and gives the proof obligations generated for each module on the different refinement levels: B, B², and B0. The table shows that the number of proof obligations are easily manageable by the automatic prover. For the example presented, the Atelier-B prover accomplished an almost automatic proof. 8 remaining proof obligations were proven by a simple manual invocation of the alternate predicate prover of Atelier-B. A single proof obligation derived from buffer at B0 could not be proven by the predicate prover and remained unproven until tackled by an interactive proof. This interactive proof and its optimisation took about 8 hours. Here, optimisation means a reduction of the number of interactive proof commands. Investigations have shown that the automatic prover failed to prove the range of a variable, the value of which was derived from a complicated arithmetic and algebraic expression.
7 Conclusion and Outlook

A novel approach to systems design has been investigated, defining the integrated tool oriented application of formal verification through model checking and formal refinement by deduction. Two tools have been applied in their corresponding domain of formal verification. The RAVEN model checker from the University of Tübingen with its input language RIL has been applied in combination with a tool for deductive refinement, namely Atelier-B from ClearSky, which implements the B language, and a B prover. The experience with RAVEN has shown, that it is a powerful tool for the verification of state based systems including timed transition. About the experience with B, it can be stated, that the language is indeed powerful in formally expressing predicates and substitutions. However, for efficiently applying B, one needs sufficient experience and education in the B methodology.

Our experiments have successfully demonstrated that model checking in combination with refinement by deduction can be successfully applied for real-time systems design, i.e., for refinement from a non deterministic, cycle accurate to a deterministic time annotated model. After testing various alternatives, we found an efficient translation from RIL to B where the generated B code can be efficiently proven by Atelier-B without any major manual interference. The mapping presented herein is based on manual translation.

In future work, we will investigate more complex models with min/max-time intervals and elaborate on the refinement of system properties so that CCTL properties are covered by the B proof.

References


<table>
<thead>
<tr>
<th>Modules/Levels</th>
<th>Obvious Proof Obligations</th>
<th>Non Obvious Proof Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>B (Structure)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>MISC</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>buffer</td>
<td>43</td>
<td>1</td>
</tr>
<tr>
<td>consumer</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>maintain</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>producer1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>producer2</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>producer3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>B (Behavior)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer</td>
<td>128</td>
<td>13</td>
</tr>
<tr>
<td>consumer</td>
<td>95</td>
<td>2</td>
</tr>
<tr>
<td>maintain</td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>producer1</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>producer2</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>producer3</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>B'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMES</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>TIMER</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>consumer</td>
<td>108</td>
<td>25</td>
</tr>
<tr>
<td>maintain</td>
<td>48</td>
<td>11</td>
</tr>
<tr>
<td>producer1</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td>producer2</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td>producer3</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td>B0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>MISC</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>TIMES</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>TIMER</td>
<td>53</td>
<td>13</td>
</tr>
<tr>
<td>buffer</td>
<td>88</td>
<td>47</td>
</tr>
<tr>
<td>consumer</td>
<td>432</td>
<td>13</td>
</tr>
<tr>
<td>maintain</td>
<td>147</td>
<td>4</td>
</tr>
<tr>
<td>producer1</td>
<td>160</td>
<td>4</td>
</tr>
<tr>
<td>producer2</td>
<td>160</td>
<td>4</td>
</tr>
<tr>
<td>producer3</td>
<td>160</td>
<td>4</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2081</td>
<td>186</td>
</tr>
</tbody>
</table>

Table 1: Proof Obligations of Different Modules