A Refinement Calculus for Promela

Asankhaya Sharma
Department of Computer Science
National University of Singapore
asankhs@comp.nus.edu.sg

***Abstract*—The use of formal methods for developing software is increasing. However in many cases only a model of the system is validated against a set of specifications. The actual implementation may thus not correspond to the formal model. One approach to this problem is to directly verify the actual implementation. Another solution is to provide a refinement scheme for the model. In this paper we present a method for refining a given Promela model to an implementation in C. We show that our refinement preserves the temporal properties (specified in LTL) of the original model. We give a new dual action semantics for a minimal core of Promela (called Featherweight Promela). The operational semantics of Featherweight Promela makes it easier to define the refinement calculus as a set of structural rules. Based on our calculus, we derive syntax directed translation rules for refinement of Promela models to C programs. These translation rules are easier to apply and generate an implementation which is a refinement of the formal model. We have applied our approach on existing Promela models and a larger case study of the cardiac pacemaker challenge.**

I. Introduction

Formal methods have been used to validate requirements and designs of software systems typically early in the development life cycle. The use of formal methods for software and hardware design is motivated by the expectation that, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems [1], [3] where safety or security is of utmost importance. Over the last several years many tools have been built that aid in formal modeling and model checking of software. But these tools typically work on a model or specification of the system. Several process based languages (Z, CSP, Event-B, Promela, etc.) have been designed to specify such formal models.

Promela is the language in which models are specified for the SPIN model checker [9]. SPIN is a popular model checking tool which aids in verification of software and hardware systems. Users can specify temporal properties of the system as LTL formulas and the tool validates if the Promela model satisfies the property. The implementation may then be based on the model and written in a language like C. However this step is usually ad-hoc and requires programmer ingenuity to ensure that the original properties of the model are captured by the implementation.

A formal model written in a specification language like Promela is meant to aid in reasoning about the program. It is not directly possible to use it for execution as it may contain constructs like non-determinism, channel based message passing and concurrency. While an executable program in a language like C (based on the model) may fail to capture all the necessary properties of the model. We have identified this gap between a formal model in Promela and its implementation in C. In this paper we propose a refinement calculus which enables users to refine a Promela model into a C program which preserves all the original LTL properties of the model. This makes it possible to generate implementation from a Promela model. We have applied our approach in the context of formal development for the pacemaker challenge [22]. As an example, consider the following Promela code snippet taken from [1]. It shows a particular operating mode of the cardiac pacemaker model.

```
if
::((timer - lastpulsev) > mintime &&
   (timer - lastpulsev) < maxtime ) ->
   pulv = 1;
   lastpulsev = timer;
   avdelay = -1;
::else->skip;
fi;
```

The model checks if the time since the $lastpulsev$ is between $mintime$ and $maxtime$, and then sets $pulv$ to 1 sending a new pulse to the corresponding chamber of heart (ventricles). We want to ensure that between successive pulses sent to heart by the pacemaker there is always a $mintime$ gap. This can be specified in LTL as the safety property $G((pulv = 1) \Rightarrow (timer - lastpulsev) > mintime))$ (where G stands for Globally). The SPIN model checker can then be used to validate if the model satisfies this property.

Our refinement calculus (described in section III) can generate an implementation based on this model. We also guarantee that the implementation will preserve the original LTL properties proven for the model. By using the syntax directed rules of section IV we get the following C code for the above mentioned snippet.

```
if(((timer - lastpulsev) > mintime &&
   (timer - lastpulsev) < maxtime )) {
          pulv = 1;
          lastpulsev = timer;
          avdelay = -1; }
else ;
```

This simple example requires application of only a few rules and the generated code looks similar to the model. The only difference is with regards to the non-deterministic choice operator, in this case it can simply be replaced with a conditional statement in C. In addition to non-determinism, some of other challenges in building a refinement calculus for

Promela include concurrency, channel based communication and under specification of the model. Due to lack of a formal semantics for Promela it is hard to show that the refinement will preserve the LTL properties.

We address these issues in this paper by proposing a novel dual action semantics for an operational core subset of Promela language (called Featherweight Promela). This semantics makes it easier to reason about the language and prove that the refinement calculus preserves temporal properties of the model. In particular our main contributions are

- **Featherweight Promela (FP)** a core subset of Promela which is easy to formalize with well defined syntax and semantics.
- A novel **dual action operational semantics**, which helps to isolate the concurrency in a language by using a Global Semantics which instantiates the Local Semantics. We describe this semantics for FP and an imperative C like language.
- A **calculus for refinement** of FP based on structural rules which ensure that temporal properties are preserved.
- **Syntax directed translation rules** from Promela to C. These rules are used to implement the refinement calculus directly for the full Promela language and generate executable C code.
- A **real time extension of FP** with clocks and timing constraints and its refinement to generate implementations for real time systems.

In the next section we introduce FP and describe the dual action semantics of the language. In section III we present our calculus of refinement. In section IV we discuss the implementation of the refinement calculus from Promela to C. We present the real time extension of our refinement calculus in Section V. Section VI shows how our approach can be applied to existing Promela models. Section VII surveys some related work and finally we conclude in section VIII.

prog ::= p*

p ::= t id (t x)* { e }

e ::= x | t x ';' e | x ':=' e | $e1$ ';' $e2$
| '::' be '→' e | 'if' e 'fi' | 'do' e 'od'
| $e1$ '!' $e2$ | $e1$ '?' $e2$ | 'run' p | 'atomic' e

be ::= ae | $be1$ '&&' $be2$ | $be1$ '||' $be2$
| $be1$ \$ $be2$ (\$ ∈ {=, ≠, <, >, ≤, ≥})

ae ::= $v1 \circ v2$ ($\circ \in \{+, -, *, /\}$)

x ::= 'true' | 'false' | v | '()' (v is an integer value)

t ::= 'int' | 'chan' | 'mtype' | 'bit'

id ::= identifier

Fig. 1: Syntax of Featherweight Promela (FP)

II. Featherweight Promela (FP)

Promela is used as a specification language for the SPIN model checker. As such the existing implementation of SPIN provides a semantics for the language. In order to reason about Promela programs and develop a calculus for refinement we propose to work with a core subset of the language. Figure 1 shows the syntax of FP, it includes all essential features of Promela like channel operators (! and ?), non-deterministic choice (::) and the corresponding data types.

Each $proctype$ process of Promela is represented by p and is executed by a run command. A process in FP p is a sequence of statements e and includes the usual control structures like if conditional and do loop. In addition there are two channel operators - write ! and read ?. As an example, $e1$? $e2$ reads the contents of the channel $e1$ into $e2$. For simplicity we assume channels to be asynchronous, but we remove this restriction in section IV. The $atomic$ keyword before an expression e ensures that the expression will be executed atomically. To complete the discussion about the syntax of the FP language, we mention that it also includes boolean expressions be and simple arithmetic expressions ae.

FP is similar to the core language used in [10] for automatic symmetry detection for Promela. Since FP includes all the constructs described in the core language of [10], any program written in Promela can also be written in FP. However, our notion of semantics is quite different from the existing work. The current informal semantics for Promela uses a notion of Kripke structures and transitions between states. This transition system based approach is useful for understanding the model checking algorithm of Promela programs. In SPIN a LTL property is checked for a model by performing a search through the state space. But this makes it harder to refine such a structure to a real world programming language like C. In order to better facilitate such refinement we propose a novel dual action semantics that cleanly separates concurrency actions of the model from their sequential counterpart.

$$\text{Assign}_1 \frac{}{< x := v, s > \longrightarrow < (), s[x \mapsto v] >}$$

$$\text{Assign}_2 \frac{< e, s > \longrightarrow < e', s' >}{< x := e, s > \longrightarrow < x := e', s' >}$$

$$\text{Seq}_1 \frac{}{< (); e, s > \longrightarrow < e, s >}$$

$$\text{Seq}_2 \frac{< e1, s > \longrightarrow < e1', s' >}{< e1; e2, s > \longrightarrow < e1'; e2, s' >}$$

$$\text{Cond}_1 \frac{}{<:: true \rightarrow e, s > \longrightarrow < e, s >}$$

$$\text{Cond}_2 \frac{}{<:: false \rightarrow e, s > \longrightarrow < (), s >}$$

$$\text{Cond}_3 \frac{< e1, s > \longrightarrow < e1', s' >}{<:: e1 \rightarrow e2, s > \rightarrow <:: e1' \rightarrow e2, s' >}$$

$$\text{If} \frac{}{<\text{if} :: e1 \rightarrow e2 \text{ fi}, s > \longrightarrow <:: e1 \rightarrow e2, s' >}$$

$$\text{Do} \frac{}{<\text{do } e \text{ od}, s > \longrightarrow <\text{if } e \text{ fi; do } e \text{ od}, s >}$$

$$\text{ChanWrite} \frac{}{< e1 \ ! \ e2, s > \longrightarrow <:: e1 \rightarrow e1 := e2, s >}$$

$$\text{ChanRead} \frac{}{< e1 \ ? \ e2, s > \longrightarrow <:: e1 \rightarrow e2 := e1, s >}$$

Fig. 2: Local Semantics of Featherweight Promela (FP)

A. Dual Action Operational Semantics

A dual action operational semantics is a small step semantics which performs two actions at each step. A global action and a local action together form the single step in this semantics. Based on these two kinds of actions, we divide this semantics into Global and Local Semantics. Concurrency from the language is confined to the Global Semantics, while Local Semantics is used to describe the sequential part. Figure 2 shows the Local Semantics of the language. Essentially it is a small step operational semantics starting with expression e and state s of the program. The state transitions for the entire program are represented by $< e, s > \longrightarrow^* < (), s' >$, where the empty expression corresponds to $()$,

The small step semantics of an expression e can be understood in terms of transition between configurations $< e, s > \longrightarrow < e', s' >$, where $s, s' \in Store$ which is simply a mapping from variables to values. The rule for assignment $Assign_1$ causes the state to update with the value of the variable v. The rules for sequences, conditionals and do loop are standard. A channel write $e1$! $e2$ first evaluates $e1$ to see if it is true and then assigns $e2$ to it. This is shown in the rule $ChanWrite$ which converts a write channel command to a conditional and assignment. A similar rule is used for channel read command ($ChanRead$). This Local Semantics avoids the use of any non-deterministic construct and concurrency. We handle those aspects with a Global Semantics, which in turn instantiates this Local Semantics.

$$\text{Exp}\ \frac{< e, s > \longrightarrow < e', s' >}{< e, E, s > \longrightarrow < (), E \cup \{e'\}, s' >}$$

$$\text{AtomicExp}\ \frac{< e, s > \longrightarrow < (), s' >}{< e, E, s > \longrightarrow < (), E, s' >}\ \text{atomic(e)}$$

$$\text{NonDet}\ \frac{}{< (), E, s > \longrightarrow < e', E - e', s > (e' \in E)}$$

$$\text{FIFO}\ \frac{}{< (), E, s > \longrightarrow < e_i, E - e_i, s > (\forall e_j \in E, i \leq j)}$$

$$\text{LIFO}\ \frac{}{< (), E, s > \longrightarrow < e_i, E - e_i, s > (\forall e_j \in E, i \geq j)}$$

$$\text{RR}\ \frac{}{< (), E, s > \longrightarrow < e_i, E - e_i, s > (\forall e_j \in E, j = (i+1)\%n)}$$

Fig. 3: Global Semantics

Figure 3 depicts the Global Semantics of FP, which is parameterized by the given Local Semantics. E is the set of all executable expressions in the program. A single step in the Global Semantics for expression e makes a transition from state s to s' by instantiating the Local Semantics with the same expression e (as shown in the rule Exp). The expressions generated by the Local Semantics are then added to the set of all expressions E. Thus the state transitions can be represented by $< e, E, S > \longrightarrow^* < (), \{\}, s' >$. When the instantiating semantics is applied on the expression e we can follow any given local semantics. This amounts to choosing an executable expression from the program and then just executing it. For the atomic expression we use the $AtomicExp$ rule which actually evaluates the expression e completely to $()$ before taking another step. This ensures that the expression is executed atomically.

The choice of the next expression to execute is what we control by using the Global Semantics. This is analogous to how SPIN executes a Promela model, by choosing non-deterministically any executable statement at each step. By formalizing the Global Semantics this way, not only do we get more control on the choice of expression but also we are able to eliminate the discussion of concurrency from the Local Semantics. We now use different kinds of rules to pick an expression e to execute.

The default behavior for FP programs is captured by the $NonDet$ rule which picks an arbitrary expression e' from the set of executable expressions. Similarly we can think of other forms of scheduling for expressions like first in first out ($FIFO$), last in first out ($LIFO$) and round robin (RR). These schedules represent the interleaving of various statements in the program. By making this explicit in the Global Semantics of the program we can reason about it easily while defining the refinement calculus.

Taking into account the Local and Global semantics together, a single small step in our semantics takes the following actions $< e, s > \longrightarrow < e', s' >$ and
$< e, E, s > \longrightarrow < (), E \cup e', s' >$ Thus we take dual actions at each transition between the configurations. Note that this Global Semantics makes no assumptions on the underlying Local Semantics of the program. We may as well use any language with appropriate Local Semantics. We use this observation as a basis for the refinement calculus.

prog ::= f*

f ::= t id (t x)* { e }

e ::= x | t x ';' e | x '=' e | $e1$ ';' $e2$
| 'if' be 'then' $e1$ 'else' $e2$
| 'while' be 'do' e | 'call' f
| 'lock' x | 'unlock' x

be ::= ae | $be1$ '&&' $be2$ | $be1$ '||' $be2$
| $be1$ \$ $be2$ (\$ $\in \{=, \neq, <, >, \leq, \geq\}$)

ae ::= $v1 \circ v2$ ($\circ \in \{+, -, *, /\}$)

x ::= v | skip (v is an integer value)

t ::= 'int' | 'struct'

id ::= identifier

Fig. 4: Syntax of Core Language (C)

III. Refinement Calculus

To develop a refinement calculus for FP programs we need to define a target language. Eventually we wish to convert Promela programs into corresponding C programs. Keeping the syntax of C in mind we introduce an imperative Core Language for our calculus. Figure 4 shows the syntax of the core language.

It contains all the usual C language features like conditions, loops and functions (f). There is also a special no-op value $skip$. In addition, there are $lock$ and $unlock$ statements. They are used to model atomic blocks from FP programs. We do not include pointers or memory references in the Core Language.

Even though widely used in C, pointers are not required for handling the constructs from our source language FP. In order to handle FP data types like channels and bits, we will use the $struct$ construct in C. All the expressions have their usual meaning as in C. We now give a Local Semantics to this Core Language similar to the one in previous section.

$$\text{Assign}_1 \frac{}{< x = v, s > \longrightarrow < skip, s[x \mapsto v] >}$$

$$\text{Assign}_2 \frac{< e, s > \longrightarrow < e', s' >}{< x = e, s > \longrightarrow < x = e', s' >}$$

$$\text{Seq}_1 \frac{}{< skip; e, s > \longrightarrow < e, s >}$$

$$\text{Seq}_2 \frac{< e1, s > \longrightarrow < e1', s' >}{< e1; e2, s > \longrightarrow < e1'; e2, s' >}$$

$$\text{If}_1 \frac{}{< \text{if } v \text{ then } e1 \text{ else } e2, s > \longrightarrow < e1, s > (v \neq 0)}$$

$$\text{If}_2 \frac{}{< \text{if } 0 \text{ then } e1 \text{ else } e2, s > \longrightarrow < e2, s >}$$

$$\text{If}_3 \frac{< e, s > \longrightarrow < e', s' >}{< \text{if } e \text{ then } e1 \text{ else } e2, s > \longrightarrow < \text{if } e' \text{ then } e1 \text{ else } e2, s' >}$$

$$\text{While} \frac{}{< \text{while } e1 \text{ do } e2, s > \longrightarrow < \text{if } e1 \text{ then } e2; \text{ while } e1 \text{ do } e2 \text{ else } skip, s >}$$

Fig. 5: Local Semantics of Core Language (C)

This Local Semantics is also a small step operational semantics as shown in Figure 5. It is very similar to the one for FP given in Figure 2. All the commands have their usual meaning as in any imperative language like C. We do not discuss the semantics for Core Language in detail here as it is fairly standard and straightforward.

Looking at the similarities between the Local Semantics for FP and C, intuitively we can think of a system of rules that can refine a given FP program into a C program. Figure 6 shows a set of structural refinement rules for FP, which transform a FP construct into corresponding C construct. The rules for variables ($RefVar$) , constants ($RefConst$) and sequences ($RefSeq$) do not change anything. For now, we assume that the refinement is applied across proper data types and a corresponding data structure is already present in C. If a corresponding data type is missing we construct an appropriate type using $struct$. We lift this restriction later when we discuss our implementation in section IV. The rule for atomic expression ($RefAtomic$) refines the expression into corresponding one within $lock$ and $unlock$ statements.

The assignment construct used in C is different from FP, which is handled by $RefAssign$ rule. The control constructs in FP are refined by $RefCond$, $RefIf$ and $RefDo$ to similar ones in C. Each run statement in FP is replaced by a call to the function representing the same process as shown in $RefRun$. The syntax of C and FP are very similar, so each process p in FP is refined to a function with same name in C. Applying the rules exhaustively to a FP program refines all the expressions and commands.The refined program is now a valid C program and can be executed directly.

Thus, the structural refinement rules transform a given FP

$$\text{RefConst} \frac{[v]_{FP}}{[v]_C} \qquad \text{RefVar} \frac{[x]_{FP}}{[x]_C}$$

$$\text{RefAtomic} \frac{[atomic\ e]_{FP}}{[lock\ x; e; unlock\ x;]_C}$$

$$\text{RefAssign} \frac{[x := e]_{FP}}{[x = e]_C} \qquad \text{RefSeq} \frac{[e1; e2]_{FP}}{[e1; e2]_C}$$

$$\text{RefCond} \frac{[:: e1 \rightarrow e2]_{FP}}{[\text{if } e1 \text{ then } e2 \text{ else } skip]_C} \qquad \text{RefIf} \frac{[\text{if } e \text{ fi}]_{FP}}{[e]_C}$$

$$\text{RefDo} \frac{[\text{do } e \text{ od}]_{FP}}{[\text{while } (e \neq skip) \text{ do } skip]_C} \qquad \text{RefRun} \frac{[\text{run } p]_{FP}}{[\text{call } p]_C}$$

Fig. 6: Structural Refinement Rules

program by refining each construct in FP to a corresponding construct in C. This refinement is based only on the structure of the program and does not talk about the semantics of each expression. The original FP program may satisfy some LTL (temporal) properties which we want to preserve during the refinement.

We will now show that the rules given in Figure 6 do indeed preserve the LTL properties across refinement. In particular we prove two things, firstly, using Local Semantics we show that the set of states in the refined C program is a subset of the set of states in the original FP program. Secondly, using Global Semantics we show that the interleaving of the expressions in refined C program is a subset of the interleaving of the expressions in the original FP program. While constructing the Global Semantics (Figure 3) we made no assumptions about the underlying Local Semantics so we can use the same for both FP and C. LTL properties are defined on traces which represent all possible interleaving of the statements in the language. Showing the above will prove that the satisfiability of LTL formulas is preserved during refinement.

A. Temporal Property Preservation

Definition 1: (Set of States) A program state is a map from variables to values. Let the set of states in the original FP program be S_{FP} and the set of states after applying the structural refinement rules be S_C.

Lemma 1: The set of states in the C program is a subsest of the orignial FP program, i.e. $S_C \subseteq S_{FP}$.

Proof: The proof is by structural induction on the refinement rules. Consider the rules for variables $RefVar$ and constants $RefConst$, as per the Local Semantics of Figure 2 and 5 the refinement from FP to C changes the state from s to the same state $s[x \mapsto v]$. Thus for this case $S_C \subseteq S_{FP}$. Similarly, we can show the same for $RefAssign$ and $RefSeq$ rules. For the $RefAtomic$ rule, the $lock$ and $unlock$ statements do not change the expression inside and are just used to choose the appropriate global semantics corresponding to atomic block.

For the $RefCond$ rule, there are three cases in the Local Semantics of FP $Cond_1$, $Cond_2$ and $Cond_3$. These cases correspond to the If_1, If_2 and If_3 rules of the Local Semantics of C respectively. The only difference is the $skip$ expression

added in If_3, but since it is just a no-op the state s' is same, both in C and FP. Consequently, $S_C \subseteq S_{FP}$ after applying this rule as well.

The if rule $RefIf$ and do loop rule $RefDo$ are based on the conditional rule $RefCond$ and hence we have $S_C \subseteq S_{FP}$ for these cases as well. ■

As shown above, after applying any of the structural rules we have $S_C \subseteq S_{FP}$. The set of states in the refined C program are all contained in the original FP program. As none of the refinement rules change the value of the expressions the state invariants are not changed at all. Thus it is sufficient to show that the number of states in the refined program are a subset of the original program. This proof is based only on the Local Semantics, next we will extend this to the Global Semantics by choosing different scheduling of expressions.

Definition 2: (Set of Traces) A trace is a sequence of program states. Let T_{FP} be the set representing the possible interleaving of executable expressions as defined in the Global Semantics for the original FP program and T_C be the same for the refined C program.

As mentioned in previous section by default we assume a $NonDet$ Global Semantics for FP programs. Nondeterministic interleaving of statements is also used in SPIN to model check the temporal property and represents an exhaustive search over the total state space. Thus, a LTL property is satisfied by the FP program for all possible scheduling of expressions. In order to show that the refined C program preserves the temporal properties, we only need to show that $T_C \subseteq T_{FP}$.

This can be done by choosing the right scheduler and concurrency mechanism for C programs. In our implementation we choose pthreads library for scheduling each function in the C program in a different thread. POSIX standard [12] supports two possible scheduling for pthreads - SCHED_FIFO and SCHED_RR. We already support these in our Global Semantics as shown in Figure 3. The POSIX library also supports mutex locks which are used by the C program for atomic blocks and the Global Semantics already handles atomic expressions.

By using pthreads we would be executing the refined C program according to the Global Semantics specified by the $FIFO$ (or RR) rule. Since the interleaving generated by these scheduling are a subset of the $NonDet$ (as it contains all possible schedules) we have $T_C \subseteq T_{FP}$. Combining with the proof for Local Semantics, we have $S_C \subseteq S_{FP}$ and $T_C \subseteq T_{FP}$.

Restating the result in terms of states and transitions, we see that the refined C program has less number of states and each of the transitions on those states is actually a subset of original number of transitions. Given a FP model and the set of traces for the model T_{FP}, the FP model satisfies a LTL formula if all the traces satisfy the LTL formula. As shown above $T_C \subseteq T_{FP}$, hence a LTL property was satisfied by the FP model it will be satisfied by the C program. Hence, temporal properties are preserved by the refinement calculus. We state this result as the following soundness theorem for the refinement calculus.

Theorem 1: (Soundness of Refinement Calculus) The refinement calculus is sound with respect to LTL safety properties.

Proof: Follows from Lemma 1 and choosing the dual action constructively in the Global Semantics as above. ■

Theorem 2: (Soundness Condition for Liveness) In addition to Theorem 1, If $S_{FP} \subseteq S_C$ then the refinement calculus is sound with respect to LTL safety and liveness properties.

Proof: Since the number of states is same in both the FP model and the C program, the liveness properties are also preserved. ■

IV. Implementation

The refinement calculus described so far is based on FP and it is not clear how to implement it directly. For the refined C program to be executable directly we also need to take care of the proper syntax requirements of the language. In order to lift the restrictions of the core language, we describe a set of syntax directed rules based on the calculus. These rules help to directly refine a Promela model to an implementation in C language (with pthreads). These rules are based on the Calculus described in previous section (Figure 6). We divide them into two categories - Data Refinement ($D1$ to $D8$) and Control Refinement ($C1$ to $C8$).

So far we have avoided the issue of mismatch of data types between Promela and C. We also did not consider asynchronous channels and selection construct. In this section we lift all these limitations. The rules shown in Figure 7 handle all data and control constructs in Promela. Data Refinement rules translate Promela data structures to the corresponding ones in C. Most of the rules are a straightforward mapping to the appropriate data type in C. Rule $D7$ shows how to refine a $mtype$ into a series of $\#define$ declarations. This is done to reflect the semantics of the $mtype$ declaration in Promela. Rule $D8$ refines a Promela channel $chan$ to a queue which is represented as a buffered array in C of corresponding size.

For control refinement we have rules $C1$ to $C8$, which translate the appropriate control flow constructs from Promela to C. Rules $C3$ and $C4$ refine the send and receive operations on channels with enqueue and dequeue functions which act on the C array refined via rule $D8$. A non-deterministic choice in Promela is treated as under specification (or external input) of the model. We refine the non-deterministic operator to a stub function which is called to generate the choice deterministically.

This stub function takes care of the under specification of the model by allowing the user to implement external calls, user input, etc. as stubs. Quite often during the formal modeling several aspects of the system are not specified, since they may not be critical to the property of interest. Also some processes may represent external environment or inputs. By allowing stub functions we keep the generated C code extensible, so if needed these aspects can be added later. For now, to mimic the exhaustive search of the Promela model (by SPIN) we use the POSIX threads library to execute one of the satisfied branches using a thread as shown in rule $C7$. We separate each branch into a function which can be executed atomically by using a lock. We use a global variable $turn$ to

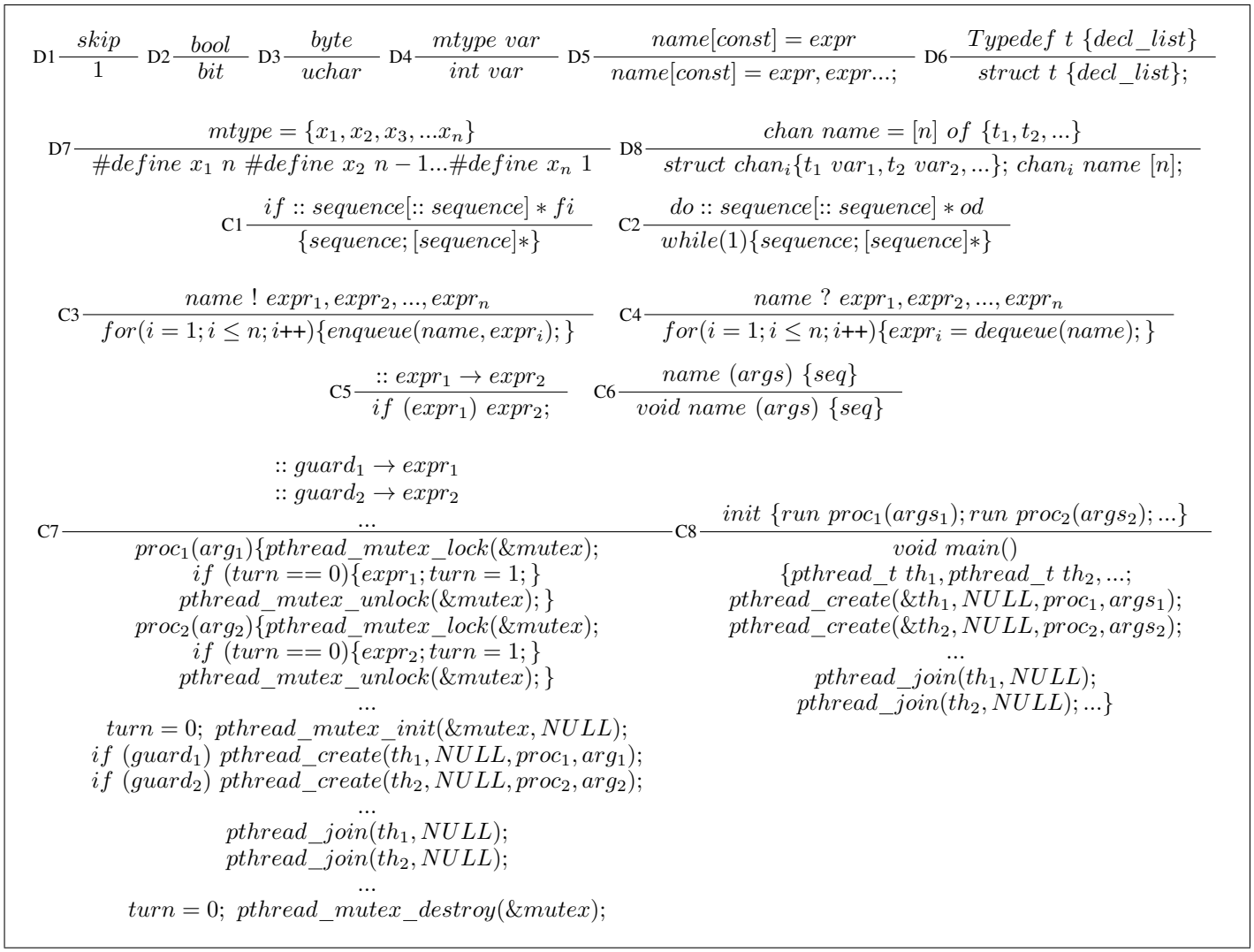

Fig. 7: Syntax Directed Refinement Rules

ensures that only one of the satisfied branch is executed. Then we create threads for each of the branch based on satisfied guards and execute them in parallel. This enables us to keep the same behavior as the model if we were to validate this C code. Note this does not change the temporal behavior of the program as all the branches are checked to validate a LTL property (by SPIN). The interleaving between various Promela processes are refined using POSIX threads. We translate each process as a C function and then create a thread to execute that function as shown in rules $C6$ and $C8$. The initial $init$ process in Promela is translated to the $main$ function in C. This set of 16 refinement rules is sufficient to translate most of the commonly used constructs in Promela to C code. These rules are similar in spirit to the ones described in [21] for refinement from B specifications to C. Applied exhaustively, these rules translate a given Promela model to C code, this code is directly executable.

In additional to these refinement rules, we present 3 synchronization rules which translate the atomic blocks and synchronous channels in Promela. As shown in Figure 8 the atomic block in Promela is refined using a global Pthread mutex lock. We lock the mutex before entering the atomic block and unlock it after exiting the block. This ensures that the changes in the atomic block are not visible to the other threads. Rules $S2$ and $S3$ handle synchronous channels using Pthread barriers. We initialize a global barrier for each synchronous channel and wait on the barrier after writing to the channel (in $S2$) and before reading from the channel (in $S3$). This ensures that the writes made to the channel are not queued and are visible to the thread reading from the channel immediately. Compare this with the rules $C3$ and $C4$ where we use a queue to handle updates to the channel. The refinement rule for the atomic block $S1$ along with rules $S2$ and $S3$ enable synchronous channels in the generated C implementation. This C code generated using the POSIX threads library behaves in the same manner as intended by the Promela model and preserves the temporal properties of the model.

A limitation of these rules is that a Promela model is a super set of all possible system behaviors, even though we can guarantee that the generated C implementation is a subset of those behaviors it may not be the desired subset. We can only say that the C implementation will satisfy LTL properties of the model. Nondeterminism in the Promela model is a big source of under specification of desired properties.

$$\text{S1}\ \frac{atomic\{expr\}}{\begin{array}{c}pthread\_mutex\_lock(\&mutex);\\ expr;\\ pthread\_mutex\_unlock(\&mutex);\end{array}}$$

$$\text{S2}\ \frac{name\ !\ expr}{\begin{array}{c}name = expr;\\ pthread\_barrier\_wait(\&barr);\end{array}} \qquad \text{S3}\ \frac{name\ ?\ expr}{\begin{array}{c}pthread\_barrier\_wait(\&barr);\\ expr = name;\end{array}}$$

Fig. 8: Additional Synchronization Rules

As discussed earlier with the $C7$ rule we allow use of stub functions in C code to determine the non-deterministic choice operator in Promela as a way to address this issue partially. In practice the generated code it is still better than having no implementation at all and developing it by hand from the model. In our experiments, we have found that the syntax directed nature of these rules make it easier to implement our calculus. These rules can be automated and applied to the Promela model as a translation to extract the refined C program. Our refinement calculus guarantees that the generated implementation will satisfy all the temporal properties of the model.

V. Real Time Extension

In one of our experiments, with the cardiac pacemaker we need the system to support real time constraints . That is, take into account the actual time elapsed between certain system events. The Promela language and LTL is not expressive enough to represent such cases. We make use of RT Promela, [25] a timed extension of Promela which adds support for real time systems. The syntax of FP is extended with a special kind of global variable for clocks and timed statements as shown in Figure 9.

prog ::= p*

p ::= t id (t x)* { e }

e ::= $untimed_e$ | $timed_e$

$timed_e$ ::= 'when' '{' u '}' $untimed_e$
| 'reset' '{' R '}' $untimed_e$
| 'when' '{' u '}' 'reset' '{' R '}' $untimed_e$

u ::= $ineq$ ',' u

R ::= c ',' R

$ineq$ ::= c \$ x | c_1 \$ c_2 '+' x (\$ $\in \{=, <, >, \leq, \geq\}$)

t ::= 'int' | 'chan' | 'mtype' | 'bit' | 'clock'

id ::= identifier

Fig. 9: Syntax of Real Time Featherweight Promela (RTFP)

The global clock type allows for declaring a new clock variable (c). All the clocks are started at 0 and are incremented in an infinite non-zeno (doesn't converge to a bounded value) time sequence. The clocks may be reset back to 0. An expression can be either timed or untimed. The untimed expressions are all the same was before in FP (from Figure 1). The new timed expressions allow expressing constraints on clock variables using the $when$ keyword. The constraints are limited to the simple linear expressions shown in $ineq$. This limitation is required to enable an efficient procedure for validating RT Promela programs as described in [25]. The constraints in the set u are equivalent to a conjunction of the individual constraints. The clocks specified in the set R can be reset using the $reset$ statement.

The RTFP programs can be verified by the RT Spin tool using a timed buchi automata (similar to SPIN). The details of the RT Spin tool are available in [25], in this paper we focus on evaluating the timed statements using our dual action semantics. The Local Semantics of the timed statements is presented in Figure 10.

$$\text{When}\ \frac{< u, s > \longrightarrow < e1, s' >}{< \text{when}\ \{u\}\ e2, s > \longrightarrow < atomic\ \{:: e1 \rightarrow e2\}, s' >}$$

$$\text{Reset}\ \frac{}{\begin{array}{c}< \text{reset}\ \{c1, c2, ...\}\ e, s > \longrightarrow\\ < atomic\ \{c1 := 0; c2 := 0; ...; e\}, s >\end{array}}$$

Fig. 10: Local Semantics of RTFP

The Local Semantics of $when$ statement checks if the constraints on the clock variables are satisfied and executes the untimed statement using the conditional expression. While $reset$ statement resets the clocks to 0 and then executes the untimed expression. The Global Semantics of RTFP are same as before, the only difference is that now the next expression to be evaluated also takes into account the constraints on clocks. The refinement for these real time expressions to corresponding C code can be supported by using the real time extensions [13] of the POSIX standard.

Figure 11 gives the rules for refinement of a Real Time Promela model. Even though the refined C code corresponding to real time rules is to be executed atomically using locks (as detailed Figure 7) we do not show it here in order to avoid repetition. The rule $RT1$ refines a global clock declaration by using the real time clock on the system and initializing a $time\_t$ variable to that value. The POSIX function $clock\_gettime$ returns the time with a resolution of at least 20 ms (as defined in the standard). However for simplicity we just show the time in seconds. The rule for clock constraints ($RT2$ and $RT3$) take into account the current time (now) and then builds the constraints for elapsed time between events.

The comma separated constraints are all combined into a single conjunction using rule $RT4$. This conjunction is used

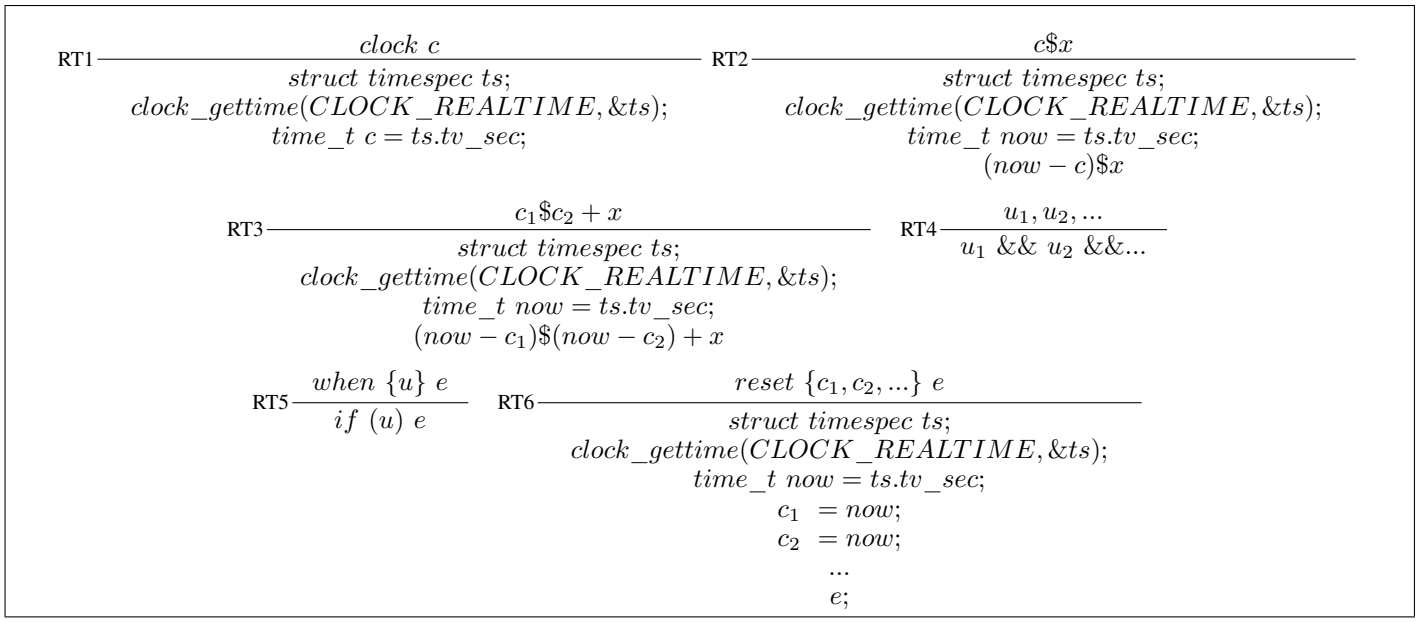

Fig. 11: Refinement Rules for Real Time Extension

in $RT5$ to refine the $when$ statement to the corresponding if statement in C implementation. The $reset$ statement is refined by rule $RT6$ by updating the current time on the clocks and then executing the untimed statement. These rules can also be applied automatically in a syntax directed manner to generate an implementation from the Real Time Promela model. The refinement of real time extension of Promela model does not change the control structure of the program. Each rule refines a RTFP expression in a state to a similar expression in the same state in C. Using the argument made in section III-A we can say that the refinement of RTFP also preserves the temporal properties. Thus the refinement calculus is sound for the real time extension as well.

VI. Experiments

In order to evaluate our approach we designed two kinds of experiments - refinement of existing Promela models and using refinement as part of formal development of a new system. We have implemented a prototype in Objective Caml called SpinR (Spin with Refinement). Source code of SpinR is made available online at [24]. A detailed example of refinement of a RT Promela model capturing mutual exclusion is given in Appendix. We have applied our calculus on a set of existing Promela models from literature [15]. These Promela models form a diverse set of case studies, which includes a real time system (Rate Scheduler), non-deterministic algorithm (8-Queens Problem) and a distributed system (Chandy Lamport's Algorithm). These models use different kinds of Promela constructs and commands which are challenging to handle for refinement. For all these case studies we downloaded the existing Promela model from [16] and applied our refinement calculus to generate the corresponding C code.

In addition to these models, we also undertook a formal development of cardiac pacemaker. The pacemaker challenge [22] is a formal development problem that models an artificial cardiac pacemaker based on the specifications issued by Boston Scientific [6]. Based on the informal specifications, we designed and modeled various components as part of the pacemaker challenge in Promela. We also formulated several temporal properties of interest for the pacemaker. Our efforts towards a verified cardiac pacemaker are described in [2], [1]. Formal model development for a critical system like the pacemaker is in itself a hard problem. Once we formulate and validate the properties on the model there is no easy way to convert that model into executable code. We used the refinement calculus described in this paper to refine the pacemaker model to C code which can then be run on supporting target hardware (pacemaker device controller).

TABLE I. Refinement with Promela Models

Promela Model	LoC	LoC(C)	States	States(C)
Sparse Channels	86	119	106	14
8-Queens Problem	50	65	41525	39225
Rate Scheduler	87	93	27	27
Fisher's Algo	60	74	494729	4337
Chandy Lamport's Algo	168	214	14632233	13063946
Pacemaker Challenge	389	431	35684919	392716

Table I shows the results of our experiments. All the test cases in the experiments with Promela models and generated C code is available online at [24]. The LoC column depicts the number of lines of code in the Promela model and $LoC(C)$ shows the same value for the generated C code. Our models range to a few hundred lines of Promela code with the pacemaker model being the largest (389). The refinement usually increases the number of lines in the program as we expand several Promela constructs while translating based on the rules of Figure 7. The increase in code size is modest for all of the case studies.

We also show the number of states in the Promela model ($States$) and compare it with the refined C program ($States(C)$). The states are calculated using SPIN as it does an exhaustive search over the state space. The scheduling used for C program corresponds to RR from our Global Semantics. We see that the refined C program has less number of states for all the cases except the Rate Scheduler (which has equal number of states). Since the refined C program has only a subset of the behaviors of the original Promela model, the

number of states in the program are reduced. Our proof for temporal property preservation was also based on this fact. In general there is a large decrease in the states space as is evident from the Pacemaker Challenge example.

Based on our experiments with several small to modest sized Promela models and one real world case study of the Pacemaker Challenge we see that the refinement calculus for Promela can be a useful component for formal modeling of systems. In particular, for new formal development efforts, this approach is easy to implement. We were able to use the refinement calculus to generate an implementation for the pacemaker software after verifying the model of the system in SPIN. Thus this calculus is an essential ingredient to make the formal development process more end to end. Our pacemaker case study is challenging and unique in this aspect as we are able to use SPIN to do verification and validation during the full development life cycle of the system.

VII. Related Work

Refinement based on Hoare style pre-post specification of programs has been studied widely in literature [18], [19], [20]. The presence of pre and post conditions helps to guide the refinement and ensures the correctness of the program at every step. However, it is not possible to express temporal properties directly in Hoare's logic for refinement. The SPIN model checker [9] enables temporal property verification (expressed as LTL formulas) of Promela models. Prior work [7] on compiling Promela models to C does not ensure that the temporal properties satisfied by the model are preserved during the translation. The authors use the existing features of SPIN to validate the generated C code directly. There is a difference between the validation of a Promlea model and C code. While validating the C code directly, SPIN treats whole functions in the program as single atomic steps [8]. This does not correspond to the non deterministic exhaustive search that happens when SPIN is validating a Promela model. Our refinement calculus translates Promela models into C programs while preserving all the temporal properties. Thus, eliminating the need to validate the generated C code again. Existing work on translating Promela into another implementation language like Java [23] also suffers from the same problem in that it doesn't respect the semantics as defined in SPIN. The refinement calculus guarantees that the dual action semantics are preserved during translation.

Other process based languages like Event-B [17] have some support for generation of implementation from models. The B-Method is a formal methodology of guided refinement based on patterns. It can be used to generate C code from models specified in Event-B. The transition system of the generated implementation needs to be shown as a Bi-Simulation of the original transition system of the model. Our refinement calculus for Promela is based on structural rules which ensures that we can generate C code for all possible models. Our work is inspired by [14], where a formal system for temporal-logic property preservation is proposed for the Z language. Unlike Z, the syntax of Promela similar to C, so by using our novel dual action semantics we are able to easily derive syntax directed rules that translate models into implementation while preserving LTL properties.

We have applied our calculus to derive an implementation for pacemaker challenge [1]. The pacemaker challenge presents a real world case study of applying formal methods for the development of various components of a cardiac pacemaker based on high level specifications [6]. Formal models for the pacemaker have been proposed and developed in [3], [4], [5]. These systems model interesting properties of the pacemaker software using languages like Z, VDM and CSP respectively. We have developed a formal model in Promela for several components of the pacemaker [1]. By using the refinement calculus presented in this paper we were also able to derive an implementation in C. This extends the current work on formal pacemaker development and makes it more end to end.

VIII. Conclusion

Model checking for temporal property verification typically works on a formal model of the system. The implementation derived from such a model may not always correspond to the original system. Thus, there is no way to ensure the correctness of the implementation. We presented a calculus for Promela, which enables to refine models written in Promela to implementation in C. Our approach is based on a novel dual action semantics, using which we show that the temporal properties proven on the model are preserved during refinement. We have applied our calculus to a modest set of existing models and to a real world case study for the pacemaker challenge. Initial results are promising and show the usefulness of the calculus for formal development.

For future work we would like to provide better tool support for the calculus by integrating it within the SPIN model checker and allowing users to extract implementations from it. This will enable to bridge the gap between formal model development and software implementation.

Acknowledgment

We would like to thank Wei-Ngan Chin, who reviewed an earlier draft of this paper and provided valuable suggestions for formalizing the approach.We are also grateful to Andreea Costea for her feedback and comments on the paper.

References

[1] A. Sharma. End to End Verification and Validation with SPIN. In CoRR 2013. http://arxiv.org/abs/1302.4796

[2] A. Sharma. Towards a Verified Cardiac Pacemaker. In NUS Technical Report 2010. http://www.comp.nus.edu.sg/~asankhs/pdf/Towards_Verified_Cardiac_Pacemaker.pdf

[3] A. O. Gomes and M. V. M. Oliveira. Formal Specification of a Cardiac Pacing System. In FM 2009.

[4] H. D. Macedo, P. G. Larsen and J. Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In FM 2008.

[5] L. A. Tuan, M. C. Zheng and Q. T. Tho. Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker. In SSIRI 2010.

[6] Pacemaker System Specification. In Boston Scientific 2007.

[7] S. Loffler. From Specification to Implementation: A PROMELA to C Compiler. In Project Report Ecole Nationale Suprieure des Tlcommunications.

[8] G. J. Holzmann. Logic Verification of ANSI-C code with SPIN. In SPIN 2000.

[9] G. J. Holzmann and R. Joshi. Model-Driven Software Verification. In SPIN 2004.

[10] A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Promela. In J. Autom. Reasoning 41(3-4): 251-293, 2008.

[11] Proposal for the Pacemaker Formal Methods Challenge hosted by SQRL. http://sqrl.mcmaster.ca/pacemaker.htm. Retrieved on 26th Sep 2011.

[12] Thread Scheduling. In IEEE POSIX 1003.1c standard.

[13] Real-time Extensions. In IEEE POSIX 1003.1b standard.

[14] J.Derrick and G. Smith. Temporal-logic property preservation under Z refinement. In Formal Aspects of Computing (25th May 2011).

[15] M. Ben-Ari. Chapter 11 Case Studies. In Principles of Spin Model Checker 2008 edition.

[16] http://www.springer.com/978-1-84628-769-5. Source of PROMELA models for case studies. Retrieved on 27th Sep 2011.

[17] J-R. Abrial. In Modeling in Event-B: System and Software Engineering. Cambridge University Press 2010.

[18] R-J. Back and J.v. Wright. Refinement Calculus A Systematic Introduction 1998 edition.

[19] R-J. Back. Correctness preserving program refinements: Proof theory and applications. Mathematical Centre Tracts, 131, 1980.

[20] E. Denney. A theory of program refinement. Technical Report ECS-LFCS-99-412, University of Edinburgh, 1999.

[21] D. Bert, S. Boulme, M-L. Potet, A. Requet and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In FME 2003.

[22] http://sqrl.mcmaster.ca/pacemaker.htm. Pacemaker Formal Methods Challenge. Retrieved on 10th Feb 2012.

[23] E. Vielvoije. SPIN's Promela to Java Compiler, with help from Stratego. In Master's Thesis 2008 TU Delft.

[24] https://github.com/codelion/SpinR. SpinR - Spin with Refinement. Retrieved on 21th Feb 2013

[25] S. Tripakis and C. Courcoubetis. Extending Promela and Spin for Real Time. In TACAS 1996.

APPENDIX

RT-Promela Model (Mutual Exclusion)

```
#define N 5 /* number of processes */
#define deltaB 1
#define deltaC 2
clock y[N];
int x,crit;
proctype P (byte id)
{do::
    reset{y[id]} x == 0 ->
    when{y[id < deltaB} reset{y[id]}
       x=id+1 ->
          atomic{when{y[id]>deltaC}
             x==id+1;
             crit++;} ->
          atomic{x=0;crit--;}
od}

init{
byte proc;
atomic{
   crit = 0;
   proc = 1;
   do
   :: proc <= N ->
        run P (proc%N);
        proc = proc + 1
   :: proc > N -> break
  od}
}
```

Refined C Implementation

```
#define N 5 /* number of processes */
#define deltaB 1
#define deltaC 2
time_t y[N];
int x,crit;
void P (int id)
{while(1){
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME,&ts);
    time_t now = ts.tv_sec;
    y[id] = now;
    if(x == 0){
      struct timespec ts;
      clock_gettime(CLOCK_REALTIME,&ts);
      time_t now = ts.tv_sec;
      if((now-y[id]) < deltaB){
       struct timespec ts;
       clock_gettime(CLOCK_REALTIME,&ts);
       time_t now = ts.tv_sec;
       y[id] = now;
       x=id+1;
       pthread_mutex_lock(&mutex);
       struct timespec ts;
       clock_gettime(CLOCK_REALTIME,&ts);
       time_t now = ts.tv_sec;
       if((now-y[id]) > deltaC){
          if(x==id+1)
             crit++;}
       pthread_mutex_unlock(&mutex);
       pthread_mutex_lock(&mutex);
       x=0;
       crit--;
       pthread_mutex_unlock(&mutex);}}}
}

void main(){
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL));
pthread_t threads[N];
int proc;
struct timespec ts;
pthread_mutex_lock(&mutex);
clock_gettime(CLOCK_REALTIME, &ts);
time_t now = ts.tv_sec;
for(int i = 0;i<=N;i++)
y[i] = now;
pthread_mutex_unlock(&mutex);
pthread_mutex_lock(&mutex);
crit = 0;
proc = 1;
while(1){
   if(proc <= N){
     int i = proc % N;
     pthread_create(&thread[i],NUL,P,i);
     proc = proc + 1;}
   if(proc > N) break;}
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
pthread_exit(NULL);
}
```