# FORMAL VERIFICATION OF EMBEDDED SYSTEM DESIGNS AT MULTIPLE LEVELS OF ABSTRACTION

*Xi Chen, Fang Chen, Harry Hsieh*
*University of California, Riverside, CA*
*E-mail: {xichen, fchen, harry}@cs.ucr.edu*

*Felice Balarin, Yosinori Watanabe*
*Cadence Berkeley Laboratories, Berkeley, CA*
*E-mail: {felice, watanabe}@cadence.com*

## Abstract

Embedded electronics today are becoming increasingly complex, which makes their design and analysis more and more difficult. An important approach to overcome the increasing complexity is to divide the system design procedure into different but interrelated stages, and represent system designs with description at different levels of abstraction. Design and analysis tools at each stages can then be more effectively applied onto the designs at particular level of abstraction. In this paper, we focus on the formal verification of embedded system designs at multiple levels of abstraction, enabled by Metropolis design environment. Based on Metropolis framework and the model checker SPIN, a translation mechanism from Metropolis design to Promela description is presented and an automatic translator is developed accordingly. We discuss the challenges and solutions in semantically translating from an object-based system design language to a procedural verification language. To demonstrate the correctness and effectiveness of our approach for formal verification, we verify properties of typical producer-consumer systems.

## 1 Introduction

As modern embedded systems become more integrated and complex, it is crucial to be able to specify the systems at multiple levels of abstraction, so that the design space can be effectively explored by successive abstractions and design decisions can be made through successive refinements [7]. Synthesis procedures can then be used to systematically transform the specification into manufactured products. The synthesis steps may include structural transformations, where behaviors are partitioned or composed, and behavior refinement, where behaviors of the design are refined through the use of constraints or implementation annotations. A formal grounding of all system representations and operations is important for understanding, analyzing, optimizing, and eventually automating procedures. In the Metropolis framework [3], designs at different levels of abstraction are represented using Metropolis Meta-Model (MMM) format. MMM can also be used to represent many different models of computation, so different high-level languages can be compiled into MMM based on their individual model semantics. Constructs in MMM are designed to facilitate the transformations and refinements between different abstraction levels. Metropolis design environment can incorporate various backend tools that allow the designers to simulate, synthesize, or verify designs specified in the MMM. A flow diagram for metropolis framework is shown in Figure 1.

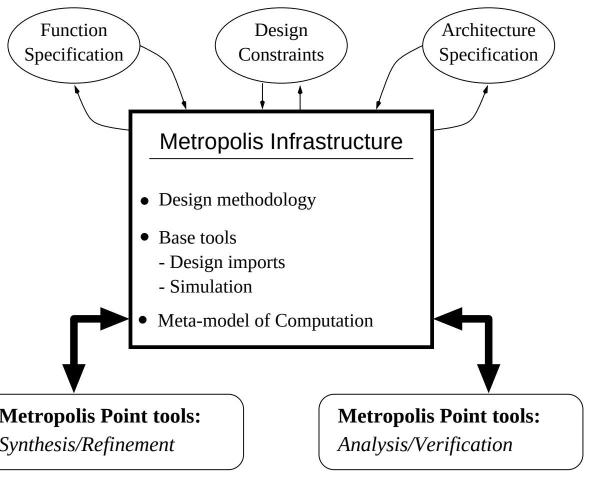
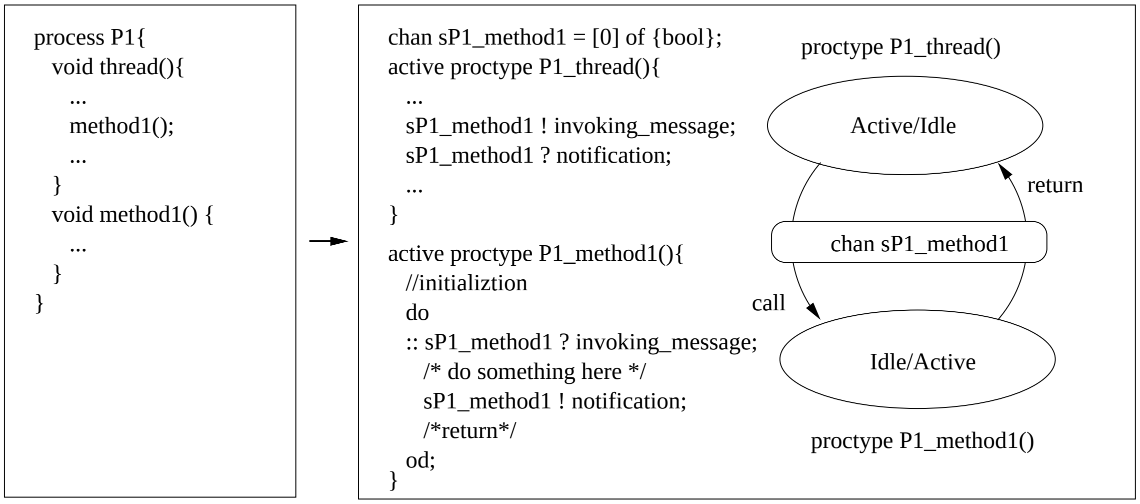


**Figure 1. Metropolis Design Framework**

In this paper, we focus on the translation mechanism from Metropolis designs to Promela, a language



```
process P1{
  void thread(){
    ...
    method1();
    ...
  }
  void method1() {
    ...
  }
}
```

```
chan sP1_method1 = [0] of {bool};
active proctype P1_thread(){
  ...
  sP1_method1 ! invoking_message;
  sP1_method1 ? notification;
  ...
}
active proctype P1_method1(){
  //initializtion
  do
  :: sP1_method1 ? invoking_message;
     /* do something here */
     sP1_method1 ! notification;
     /*return*/
  od;
}
```

**Figure 2. Translation of MMM Processes.**

suitable for formal verification. Spin [6, 1], a formal verification tool for asynchronous software systems, is chosen as a backend verification engine. Property verification is most useful at higher level of abstraction where design representation are less complex and the state space are more manageable. We concentrate on formally verifying description at or near the specification level. Verification at other abstraction levels may be accomplished through conformance testing [8], or through simulation.

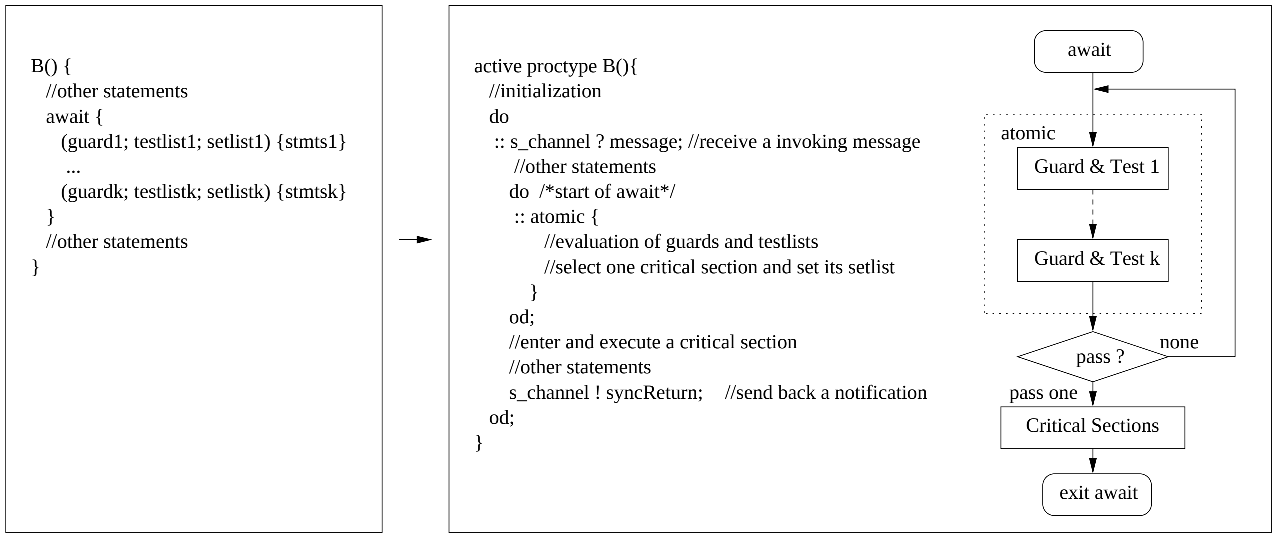
In Metropolis, designs are specified as asynchronous processes with communication specified by the medium and the overall behavior limited by the constraints and schedulers. Promela, the input language for Spin, is also an asynchronous process specification language though with much simpler communication mechanism (FIFO channels or global variables) and hierarchy . To enable verification of MMM designs, we interpret system level primitives of MMM semantically with Promela constructs and demonstrate the correctness and feasibility of the approach through a set of verification case studies.

The main challenge of this work is therefore to precisely interpret and represent the semantics of MMM designs with Promela. Our main contribution is an effective solution to enable the formal verification of MMM designs and an automatic translator. To arrive at this solution, we carefully interpret each MMM constructs and represent them with groups of Promela constructs without overly increasing the complexity. The limitations of the approach follows directly from the limitation of the model checker. When the size of a design increases, perhaps through refinement or mapping onto particular architecture, the running time and memory usage of the verifier increase exponentially. The designer will need to resort to conformance checking or simulation when resource thresholds are reached.

In the next section, we discuss specific aspects of the translation from a high-level object-oriented system specification to a procedural format [2] suitable for verification with Spin. In specific, we concentrate on the semantically correct representation of system processes, communication media, and communication primitives (i.e. the “await” statement.) In Section 3, we present a case study of property verification of a system level specification in Metropolis. In Section 4, we transform and refine the high-level specification, as would be done during synthesis, and show, through a verification case study, that our verification methodology can effectively verify designs at different levels of abstraction.

## 2 Connecting Metropolis and Spin

In this section, we briefly describe the Metropolis Meta-Model (MMM), the input representation for Metropolis design environment, and Promela, the input language for Spin model checker. We then describe the translation procedure and highlight the interesting aspects of dealing with the two semantic domains.



```
B() {
  //other statements
  await {
    (guard1; testlist1; setlist1) {stmts1}
     ...
    (guardk; testlistk; setlistk) {stmtsk}
  }
  //other statements
}
```

```
active proctype B(){
  //initialization
  do
   :: s_channel ? message; //receive a invoking message
      //other statements
     do  /*start of await*/
      :: atomic {
           //evaluation of guards and testlists
           //select one critical section and set its setlist
         }
     od;
     //enter and execute a critical section
     //other statements
     s_channel ! syncReturn;    //send back a notification
  od;
}
```

**Figure 3. Translation of *await* statements.**

## 2.1 Metropolis Meta-Model and Promela

Metropolis meta-model is a system representation formalism capable of representing designs at different levels of abstraction. In MMM, systems are represented as networks of *processes* that communicates through *media*. Processes execute concurrently, each at its own pace. The relative speed of progress of processes may arbitrarily change at any time, unless they synchronize with each other using the synchronization primitive called *await*, or some *constraints* are placed on system executions. Similar to Java or C++ objects, processes and media have member functions and member variables. Processes communicate to each other and the environment through execution of member functions implemented in the shared media. Figure 4 shows an example of processes communicating through a medium.

Probably the single most important system-level MMM construct is the *await* statement used to establish mutually exclusive sections and synchronize processes. An *await* statement contains one or more statements called *critical sections,* each controlled by a triple (*guard*; *testlist*; *setlist*), where *guard* is a Boolean expression, and *testlist* and *setlist* denote sets of interface functions of other processes. A critical section is said to be *enabled* if its *guard* is true, and none of the interface functions in the *testlist* are being executed at that moment. A critical section may start executing only if it is enabled. In addition, while the critical section is being executed, no interface functions included in the *setlist* can begin their executions. Whenever an *await* is encountered in the execution flow, one and only one of the enabled critical sections is executed. If no critical section is enabled, the execution blocks. If more than one critical sections are enabled, the choice is non-deterministic.

Promela is a procedural language suitable for formal verification. Each thread of execution is also modeled by a process. Processes may communicate either through a fixed size FIFO buffer, or shared memory variables. Coordination between processes are accomplished through the use of *atomic* statement, where if one process entered the *atomic* section, all other processes stop execution until the given process exits the *atomic* section. To avoid deadlock, the semantic of Promela dictates that if an execution is ever blocked waiting inside an *atomic* section, the atomic property is nullified and all other processes are allow to resume their execution immediately. Dynamic creation and destruction of threads are supported within Promela. However, due to efficiency consideration, the total number of processes are limited to 256. This, coupled with very conservative process destruction mechanism, renders the feature very limited in its usage.

## 2.2 Translation Mechanism

Two main issues in the translation from MMM to Promela are modeling of MMM communication mechanisms in Promela and modeling of constructs specific to MMM. The former is an issue because the semantics of the communication can be more general than those supported in Promela.

In MMM, communication between processes is made by calling functions defined in media. There are two ways of modeling function calls in Promela. One is to directly model it by inlining the called functions, where the callers are modeled as active process of Promela. Another way is to use active processes to model all instances of meta-model functions, which includes all the member functions of processes, media and other objects in the meta-model. Figure 2 illustrates this approach. Each member function is translated into an active Promela process, a process that is instantiated and initiated at the very beginning of the execution, and a function call in MMM is translated as invoking an execution of the corresponding Promela process (e.g., see thread() and method1() in Figure 2). The invocation is accomplished through message passing using a rendezvous channel (i.e. FIFO channel of size 0). Figure 2 shows the function thread() of process P1 makes a function call to method1(). In Promela, this is interpreted as process P1_thread sending a message to process P1_method1 through a rendezvous channel sP1_method1 (using operator !). The function return follows the same paradigm. When P1_method1 finishes, it sends back a notification message through the same channel to P1_thread. P1_thread receives the message(using operator ?) and continues its execution. Thus, the sequential execution flows and control transfers of the MMM processes are assured. Due to Spin's limitation on the number of running processes and its resource recycling mechanism [1], dynamically creating new processes is prohibitively expensive. Instead, all Promela processes, except the processes representing meta-model constructors and threads, are initialized at the beginning of execution as active processes blocked waiting for a invoking message from their calling processes through the corresponding rendezvous channels. Member variables of a MMM process or medium are represented by global variables of Promela after they are renamed appropriately.

In MMM, an interface is used to define the I/O data ports of the process or medium and the I/O control points of the process or medium. To implement the control point, the MMM interface is used as an integer semaphore in the *setlist* and *testlist* of an *await* statement. We translate each interface into a pair of integer semaphores in Promela. The first semaphore, called $ACTIVE$ is used to indicate whether the interface (and its member functions) are in active state (whether they are being executed), another one called $EXCLUSIVE$ indicates whether this interface semaphore is set (i.e. whether it is included in the *setlist* of some *await* statement that is currently executing). Figure 3 illustrates how an *await* statement is translated in Promela, where Promela constructs such as *atomic*, repetition do-od and case selection if-fi are utilized to guarantee the exact semantics equivalence. We use these semaphores to signal that interface functions appearing in *testlist*'s are being executed and to prevent, when appropriate, interface functions appearing in *setlist*'s from being executed.

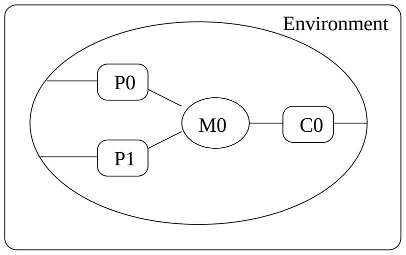


**Figure 4. The bytelink example.**

## 3 Property Checking

Property checking searches the state space of a system design and checks whether some design property holds. Spin provides two powerful methods to specify properties of a design: Assertion and Linear Temporal Logic (LTL) properties [5]. Assertion is an annotation construct in Promela that is targeted for checking local condition in a design. If an assertion condition is specified at the MMM level, it can be directly translate down to an assertion in Promela. In fact, we use this method to verify that *await* statement was correctly translated into Promela (i.e. semaphores correctly locked and released). For example, to verify that critical sections controlled by semaphores S1 and S2 are always mutually exclusive, we put the following assertion into the design itself:

$$\textit{assert( S1 == 0 || S2 == 0).}$$

And Spin model checker confirms that assertion indeed hold. Since assertion property represents a strict subset of LTL property, we will focus on the LTL property verification for the rest of this paper.

LTL property specifications allow us to specify properties at the MMM level and hence is more integral to our verification methodology. In Spin, LTLs are implemented as *never* claims. And these *never* claims are then used to monitor the execution sequence during formal verification. If no violation is found, the property of the design is verified to hold.

Let us consider a simple example written in MMM format (Figure 4). A network that includes two producer processes (P0 and P1) and one consumer process (C0). A single space data storage medium (M0) is used for communication. The producers and consumers may perform arbitrary operation to the data. Our system level design focus on how these the communication and coordination are accomplished so in fact the exact data value may be abstracted away at this level. We want to check the property: "Whenever the producer start to write an item into the medium, there must be some space in the medium". The property can be expressed as:

$$\textit{Globally( (P0\_write or P1\_write)} \rightarrow \textit{M0\_empty ).}$$

The property is proven to hold under one minute of CPU time.

Another property we want to check is: "when consumer wants to read and there is no data in medium and neither producer has started to write, the consumer cannot finish reading until either producer starts to write". This property is expressed as:

$$\begin{gathered}\textit{Globally( (C0\_start and M0\_empty and}\\ \textit{(not P0\_start) and (not P1\_start) )} \rightarrow\\ \textit{( (not C0\_end) Until}_{strong} \textit{ (P0\_start or P1\_start) ) ).}\end{gathered}$$

where the strong until operation, A $Until_{strong}$ B, being true means that A is true all the time, or A is true until B is true. The weak version of the operator, A $Until_{weak}$ B, being true means that eventually B has to become true. This property is verified in under one minute of CPU time. All the verification parameters involved are listed in Table 1.

**Table 1. Summary of verification parameters**

| | |
|---|---|
| Depth reached | 589790 |
| State generated | 1.65642e+06 |
| State transitions | 6.34532e+06 |
| Total memory used | 317.251 MB (Partial Order Reduction) |
| CPU time elapsed | 43.79s |
| CPU type | Athlon 1.5GHz |

## 4 Transformation and Refinement

Synthesis procedures transform and refine the specification and produce a detailed description of the implementation. MMM can be used to formally represent the design before and after a particular synthesis step. We describe an example where such a synthesis step is taken and show how properties can be verified at different levels of abstraction.

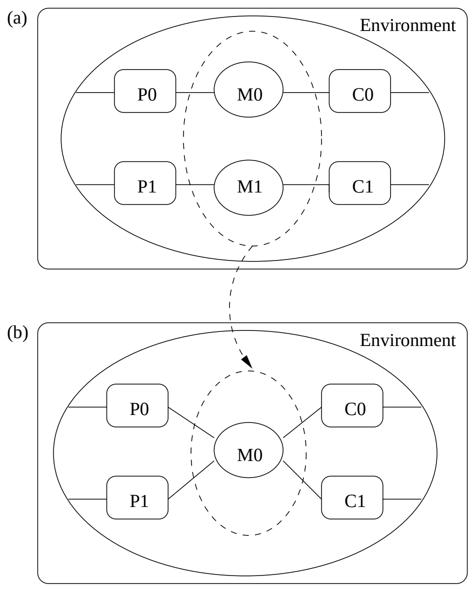


**Figure 5. Example of a refinement.**

Consider Figure 5. In (a), two producer-consumer data streams are running independently and two single space medium are used for each to provide storage. It is straight-forward to verify using Spin that

$$\begin{gathered}\textit{Globally( (C0\_start and M0\_empty and}\\ \textit{(not P0\_start) )} \rightarrow\\ \textit{( (not C0\_end) Until}_{strong} \textit{ P0\_start ) ).}\end{gathered}$$

Now, consider (b) where a single medium is used. SPIN verifies that the property does not hold. An error trace is produced by Spin when a verification fails which includes each step with which the system executes until a violation is encountered. Analyzing the error trace allows one to catch and fix the undesired behaviors of an incorrect design. The error trace shows clearly that for the property to hold, extra constraints must be added such that the two streams of data do not mix. MMM format provides constructs for constraints specification also in LTL:

$$\begin{gathered}\textit{Globally( P0\_write} \rightarrow \textit{(not C1\_read) Until}_{strong}\\ \textit{C0\_read)}\end{gathered}$$

and

$$\textit{Globally( P1\_write} \rightarrow \textit{(not C0\_read) Until}_{strong} \textit{ C1\_read)} .$$

Such constraints can be guaranteed by a correct scheduling algorithm (e.g. Round robin scheduling with P0,C0,P1,C1 ordering).

With these constraints, the property may be verified by Spin using the LTL formula of the form

$$\textit{Constraints} \rightarrow \textit{Property} .$$

The verification completes without error. However, due to the increased state space, it takes more than 9 hours on a 1.5GHz Athlon machine with 1GByte of memory. Table 2 lists the detailed resource usage of this verification. We have obviously reached the limit with this approach. Work is underway where the translator automatically inlines the active processes whenever possible. Preliminary results are very promising.

**Table 2. Summary of verification parameters**

| | |
|---|---|
| Depth reached | 13224645 |
| State generated | 7.27328e+07 |
| State transitions | 9.40072e+08 |
| Total memory used | 464.541 MB (Graph Encoding) |
| CPU time elapsed | 9h:44m:48s |
| CPU type | Athlon 1.5GHz |

## 5 Conclusion and Future Directions

In this paper, we present a verification approach for embedded system designs. This approach is unique in that it is able to operate at different levels of abstraction, enabled by Metropolis framework. Integral to the approach is a semantically correct translator from system level language, Metropolis Meta-Model, to verification/protocol language, Promela. Case studies have been performed to show the power of such an approach both in terms of property verification and formal verification of designs before and after a synthesis step.

There is obviously a trade-off between invoking process dynamically and in-lining the methods into the process whenever we could. We are currently conducting experiment to better understand this trade-off. Preliminary result indicate that the verification limit of the previous section can be ameliorated. We are also working on integrating more system level constructs into the translator, including scheduler and quantitative logic constraints [4]. Another avenue of future research is in the formal verification of platform architecture, where the property of a platform, as oppose to the property of the design, will be verification. This will enable the formal understanding, analysis, and verification of the embedded system design from high level specification all the way down to low level implementation.

## References

[1] Spin manual, http://netlib.bell-labs.com/netlib/spin /whatispin.html.

[2] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. *Technical Report 2001/01 Cadence Berkeley Laboratories*, Nov. 2001.

[4] F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. *International Workshop on High Level Design Validation and Test - HLDVT01*, Sept. 2001.

[5] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. *Proc. IFIP/WG6.1 Symp. on Protocols Specification, Testing, and Verification*, June 1993.

[6] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.

[7] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, Dec. 2000.

[8] J. Tretmans and A. Belinfante. Automatic testing with formal methods.