

Case Studies of Model Checking for Embedded System Designs

Xi Chen, Harry Hsieh
University of California at Riverside
Riverside, CA 92521, USA
{xichen, harry}@cs.ucr.edu

Felice Balarin, Yosinori Watanabe
Cadence Berkeley Laboratories
Berkeley, CA 94704, USA
{felice, watanabe}@cadence.com

Abstract

As modern embedded systems become more integrated and complex, it is crucial to be able to represent systems at multiple levels of abstraction, so that the design space can be effectively explored by successive refinements and abstractions. In this paper, we present a formal verification methodology and case studies for property verification of designs represented at different abstraction levels. Utilizing Metropolis meta-model (MMM), Y-chart Application Programmer's Interface (YAPI), an automatic translator, and the model checker SPIN, we verify properties for both system level representations and refined representations.

1 Introduction

Electronic products today demand high performance, high integration, and a long list of features. As a consequence, embedded electronics are becoming more complex and difficult to design. To combat complexity and explore design space effectively, it is necessary to represent systems at multiple levels of abstraction. Initial function and architecture specifications should be done at high abstraction levels, so as not to bias unnecessarily toward any particular implementation [12](see Figure 1). Through successive refinements and abstractions, the design space can be explored effectively and the design decisions can be made intelligently. Synthesis (i.e. steps taken toward implementation) is applied systematically to transform the specification into manufactured products. Synthesis steps may include structural transformations, where the design is partitioned, composed, or otherwise altered; formal refinements, where possible behaviors of the design are formally refined through the use of constraints or implementation annotations; and mapping, where functional specification at a particular abstraction level is mapped to an architectural specification at a particular abstraction level. A formal grounding for all system representations and operations is essential for the

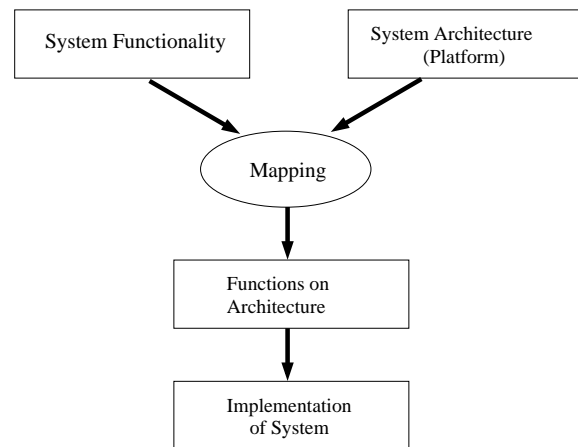


Figure 1. System Design Methodology.

ability to perform analysis, optimization, and automation.

Metropolis [4] design framework enables designers to represent and manipulate their designs at multiple levels of abstraction and with multiple models of computation (MoC). Central to the framework is the Metropolis Meta-Model (MMM) representation. Different high-level languages, models of computation, design constraints, as well as specifications of architecture platforms can be represented in MMM while retaining their correct semantics. Constructs in MMM are designed to facilitate the transformations and refinements between different abstraction levels. Incorporated into the Metropolis design environment is a set of backend tools, with which one can simulate, synthesize, and verify the design at hand. A flow diagram for metropolis framework is shown in Figure 2. In Metropolis, designs are specified as networks of processes with communication specified by media and the overall behavior limited by the application of constraints, scheduling, and mapping. At the purely functional level, processes are unconstrained and all interleaving are possible, as long as they

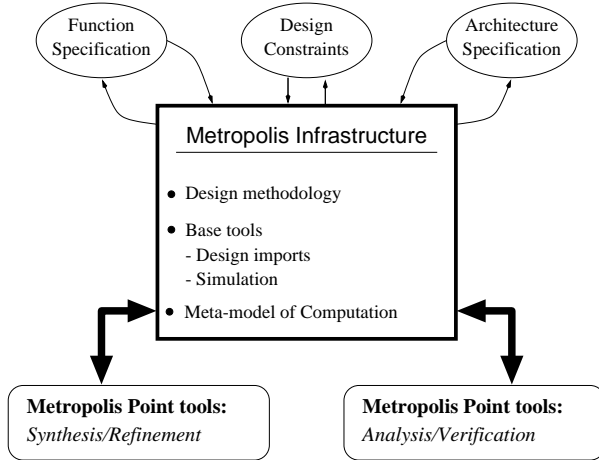


Figure 2. Metropolis Design Framework

satisfy data dependencies. A structural transformation, formal refinement, or mapping may be applied corresponding to a synthesis step. A structural transformation may be applied where media/processes are combined or split. In this case, the property of the design may not be preserved. A formal refinement may be applied where a constraint or a scheduler is defined to restrict the possible behaviors of the design. Some properties of the design (e.g. safety properties) may be preserved while others (e.g. liveness property) may not. Mapping occurs when the functional specification is mapped to the architectural specification.

Formal property verification can be very powerful for catching errors early in the design process. Formal verification tools, notably model checkers [9, 2, 15] have been made available to the designers. The designers can describe their designs with the given formal language and the properties they want to check in some logic. If a property is found to be false for the design, an error trace is provided by the model checker to the designers to help them modify the design or the property. The state explosion problem restricts the usefulness of exhaustive proving to protocols or other higher levels of abstraction. Approximate verification (e.g. the bitstate technique, an option available in SPIN [10]) allows the model checkers to automatically check the property with only a portion of the state space explored. Obviously, the approximate verification does not prove that the property holds for all conditions. The tool provides the estimated percentage of this partial exploration (i.e. confidence factor) and reports a bug if one is found on the partial state space searched.

One problem for formal property verification is that a verification model needs to be written, often manually, from the specification model. This tedious process multiplies if the designers wish to verify the properties of a design as

it goes through various abstraction/refinement operations. Metropolis, with its formal semantics, allows full integration of formal verification tools [6]. Verification models can be automatically generated for all levels of the design so the designers no longer have to manually re-describe their design in a formal verification language as the design moves from high level of abstraction toward implementation. The central challenge in this approach is that the verification languages, such as Promela used by SPIN model checker [9], allow only simple concurrency modeling and are not amenable to the system design specification where complex synchronization and architecture constraints are needed. Our translator automatically constructs the verification model from the specification model, taking care of all the system level constructs. This paper extends the groundwork laid down by [6] by defining translation algorithms for advanced constructs such as function inlining and dynamic objects, presenting a complete verification methodology, and carrying out case studies to demonstrate numerous aspects of the verifications before and after synthesis. While we focus on the verification methodology for Metropolis designs, the same approach can be easily applied to other abstraction/refinement design frameworks. Metropolis allows different Models of Computation (MoC) to be chosen when describing a design. Our methodology concentrates on verification at the higher levels, so system level designs written in the formal MoC such as the popular Y-chart Application Programmers' Interface (YAPI) and its more refined counterpart, Task Transition Level(TTL) model, are good candidates for our methodology [13, 7]. We perform a verification case study for designs using YAPI library and TTL library to demonstrate formal verification of multiple levels of abstraction in an industrial setting.

In the next section, we introduce the basic constructs and semantics of MMM and discuss aspects of the translation from Metropolis Meta-Model, an object-oriented system specification language to Promela, a procedural formal software protocol format [3]. We review the basic translation mechanism, along with the explanation of more advanced concepts such as functional inlining and dynamic objects. The designer needs to work only at the system level (i.e. MMM). The constraints along with the system specification are translated automatically into the SPIN environment. In section 3, we present our verification methodology and show how verification can be done along with the synthesis procedures. In section 4, we present case studies of the property verification of system-level specifications. The first case study we use is a prototypical scenario of m -producers, n -consumer communicating through a medium. We show how verification can be done in the presence of constraints and schedulers and how property verification can be performed before and after the composition procedure. The second case study deals with designs based on

YAPI and TTL models of computation in a realistic industrial setting. Section 5 concludes the paper and gives our future research directions.

2 MMM and Promela

In this section, we briefly review the syntactic and semantic features of Metropolis Meta-Model and Promela. Despite the similarity between the two languages, MMM contains many system-level constructs which need to be translated carefully into semantically equivalent code segments in Promela. A functioning translator has been built and its effectiveness has been tested.

2.1 MMM Format

In Metropolis, systems are represented as networks of *processes* that communicate through *media* [4]. The syntax of MMM is similar to Java but includes many system level extensions. A process is an active object and always defines a function called *thread* as the top-level function where its behavior is specified. A communication medium implements a set of functions which are grouped into *interfaces*. Processes connect to media through *ports*. Each port has a type, which must be an interface implemented by the medium to which the port is connected.

Processes run concurrently, each at its own pace. The relative speed of processes may arbitrarily change at any time, unless they synchronize with each other using the synchronization primitive (*await* statement), or some *constraints* are placed on system executions. The *await* statement can be used to make a process wait until some conditions hold and establish critical sections that guarantee mutual exclusion among different processes. To limit the behavior of processes, a designer can also put high-level LTL (Linear Temporal Logic) [14] or LOC (Logic of Constraints) [5] constraints on the system specification without giving any specific scheduling algorithm, and leave the implementation to the lower-level abstraction.

A scheduler is a special object of MMM that can be used to implement user-defined scheduling algorithms to control the coordination of a set of processes in an MMM network. Schedulers connect to these processes through *statemedia*, a special type of media. Unlike processes, schedulers are not active objects, which means that they implement functions that are called whenever scheduling needs to be done, but don't have their own running threads. With a properly designed scheduler, the running processes are synchronized in a particular way (e.g. round-robin or priority-based) to satisfy the constraints. We are currently working on the translation of other architectural platform constructs within MMM, such as the concept of architectural services and

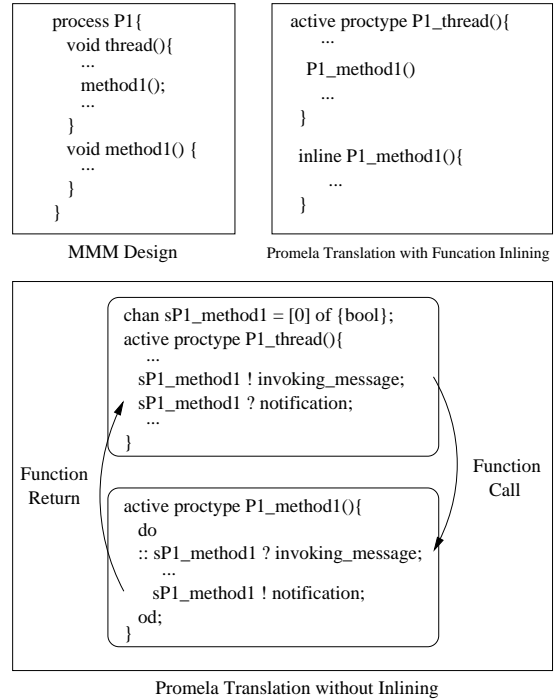


Figure 3. Translations of MMM Functions.

quantity managers. Their operations and semantics are outside the scope of this paper.

2.2 SPIN and Promela

In SPIN [9, 1], the protocol designs are specified by Promela, a C-style procedural language with a few protocol level extensions. The basic concurrent execution units are processes, which are defined with the keyword *proctype*. A communication construct, *channel*, is used to implement both synchronous and asynchronous message passing between processes (with *receive* operator *?* and *send* operator *!*). The processes may also exchange information through global variables. A useful Promela primitive is *atomic*, where only one among all the atomic blocks can be executed at any given time. To avoid deadlock, Promela's atomic statements are not blockable (e.g. it can not wait on an empty channel). MMM allows a more complex communication scheme which will be explained later in this section. To reduce the the usage of CPU time and memory space during verification, SPIN also provides several options for memory reduction (e.g. partial order reduction [11] and graph encoding [9]) and approximate verification (e.g. bit-state [10] and hash-compact [16]).

2.3 Translation from MMM to Promela

To formally verify an MMM design with SPIN, the MMM specification needs to be translated to Promela description. The main constructs of MMM are processes, media, interfaces, schedulers and await statements. Each member function of an MMM process or medium is translated to an active Promela process and a function call in MMM is translated as invoking an execution of the corresponding Promela process. The invocation is accomplished through synchronous message passing using a rendezvous channel. If we have a process called $P1$ and it has a member function called $method1()$, which is invoked in the process (see Figure 3), the function $method1()$ is translated to an active process $P1_method1$ and the invocation is implemented as message passing through a rendezvous channel ($sP1_method1$). To reduce the overall complexity of the verification, we provide a compilation option that inlines all the functions into the process that calls them directly or indirectly. To perform functional inlining, the function $P1_method1()$ is declared using the keyword *inline*. The translator simply pastes its translated code into the point of the invocation in the calling process (see Figure 3). No process or channel is needed for such a function. In the situation of multiple level function calls, all the functions are inlined recursively so that one MMM process corresponds to only one Promela process. Thus the total number of Promela processes can be shrunk, which reduces the inherent complexity of the Promela program accordingly. With functional inlining, the verification becomes much more efficient regarding both time and memory usage. To translate an await statement, the Promela constructs such as *atomic*, repetition *do-od* and case selection *if-fi* are utilized to guarantee the exact semantic equivalence. Other MMM constructs are translated to Promela similarly [6].

Another interesting aspect of MMM is the dynamic object (i.e. the reference type). For example, an array is a reference type in MMM, and its memory space could be allocated and changed dynamically at runtime. However, most model checkers (including SPIN) only support static memory allocation, i.e. arrays have to be declared explicitly at design time. To solve the problem, we have to put some restrictions on the MMM code at hand. All the reference types have to be declared explicitly once and only once, so that they can be translated to Promela as static objects. If we have an array declaration in MMM, “*int[] a = new int[12];*”, it can be translated to Promela as a static array “*int a[12];*”. After the array a is declared in MMM, its reference cannot be changed any more. If the dimension of the MMM array is dynamic, e.g. “*int[] a = new int [n];*” where n is a variable, it is also translated to Promela as a static array “*int a[ARRAY_MAX];*”, where $ARRAY_MAX$ is a constant set by the designer at com-

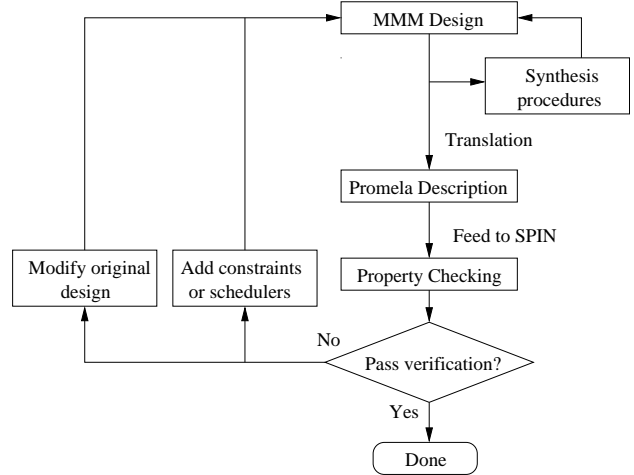


Figure 4. Formal verification methodology.

pilation time. It is up to the designer to guarantee that $ARRAY_MAX$ is always larger than or equal to the maximum value of n . Other dynamic objects such as class types in MMM are similarly translated to static data objects of Promela.

3 Formal Verification Methodology

By using an automatic translation procedure to generate verification model from system specification model, we allow designers to perform verification at different levels of abstraction as a design goes through various synthesis steps without the tedious and error-prone step of manually re-writing the design. Furthermore, the verification can be used to drive the refinement and transformation of system representation, i.e. the synthesis procedure.

SPIN provides two powerful ways to specify properties of a design: Assertion and Linear Temporal Logic (LTL) [8, 14]. Assertion is an annotation construct in Promela used to “assert” that a particular condition (e.g. $space > 3$) must hold. Assertion written with variables in MMM can be easily translated into Promela segment and inserted into the translated Promela code. The LTL formulas are constructed using terms, classical boolean operators such as \neg (not), \vee (or), \wedge (and), \rightarrow (imply), and the temporal operators \square (always), \diamond (eventually) and \mathbf{U} (strong until). Terms are Boolean conditions on variables or process states, therefore LTL is strictly more expressive than Assertion properties. Without loss of generality, we will only deal with LTL formulas for the rest of the presentation.

Our verification methodology is illustrated in Figure 4. The MMM description is automatically translated into Promela description, and properties are checked using SPIN

model checker. The designer may perform any synthesis step (e.g. composition, decomposition, constraint addition, scheduler assignment) and a new Promela code can be automatically generated for verifying the property. If it does not pass, the error trace may be used to help designers figure out whether the design needs to be altered. If the verification session runs too long, approximate verification can be used to explore a subset of the state space and report the probability that the property will pass. Obviously, a partial exploration can not prove that a property holds. However, it is our experience that a lot of “easy” bugs can be found within a relatively small amount of time and memory usage and if a SPIN verification session continues to run after a long time, it is highly likely that the property will eventually pass.

The same methodology can also be used for a verification-driven synthesis methodology. If the property does not pass the verification, an error trace is generated and examined. Based on the error trace, the original design may be incorrect, or refinement may be applied to the original specification for it to have the desired property. At a higher level of abstraction, constraints may be used to constrain the behavior so the property may pass. At a lower level of abstraction, schedulers which coordinate the interacting processes may be used to achieve the same result. Subsequent synthesis steps may then actually implement the schedulers on a particular platform.

4 Formal Verification Case Studies

The first set of case studies consider a prototypical network of m producers and n consumers communicating through a single medium (see Figure 5). The producers receive inputs from the environment, process the data in some way, and then output it to a medium of a single space. The consumers read in information from that medium, process it, and then output to the environment. It is possible for all producers and consumers to execute concurrently. We verify properties of the design before and after synthesis steps. The second set of case studies involves property checking of systems designed using YAPI model of computation and its more refined, TTL, counterpart.

4.1 Verification of data integrity

Given the networks of consumers and producers with one medium(see Figure 5), we want to check the property:

“Whenever a producer writes an item into the medium, there must be some space in the medium.”

The property can be expressed as:

$$\square ((P_1_write \vee \dots \vee P_m_write) \rightarrow M_1_not_full) \quad (1)$$

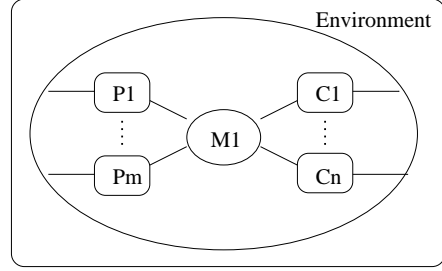


Figure 5. Example of a bytelink meta-model.

We verify the case with $m = 2$ and $n = 1$, which has 102 lines of MMM source code and 1305 lines of Promela code after translation. The property is proven by SPIN within one minute on our 1.5GHz Athlon machine with 1GByte of memory. The same setup is used for all case studies in this paper. The detailed resource usage of this verification is listed in Table 1, where the states generated are the total number of unique global system states that are stored by the verification algorithm.

Table 1. Summary of verification of property1

Depth reached	180457
States generated	439274
State transitions	2.07419e+06
Memory used	95.118 MB
CPU time elapsed	13.31s

Another property we want to check is:

“When a consumer wants to read and there is no data in the medium and none of the producers has started to write, the consumer cannot finish reading until some producer starts to write.”

In LTL, this property is expressed as:

$$\begin{aligned} & \square ((C_x_start \wedge M_1_empty \wedge \\ & \neg (P_1_start \vee \dots \vee P_m_start)) \rightarrow \\ & ((\neg C_x_end) \mathbf{U} (P_1_start \vee \dots \vee P_m_start))) \quad (2) \end{aligned}$$

where $x \in [1, n]$, and *start* indicates the condition when a consumer initiates a read operation and when a producer initiates a write operation and *end* indicates when they complete the operations. We verify the case with $m = 2$ and $n = 1$. The property is verified by SPIN within one minute of CPU time on the same machine. All relevant verification parameters are listed in Table 2.

Finally, we want the following property:

“If the consumers are able to keep reading data from the

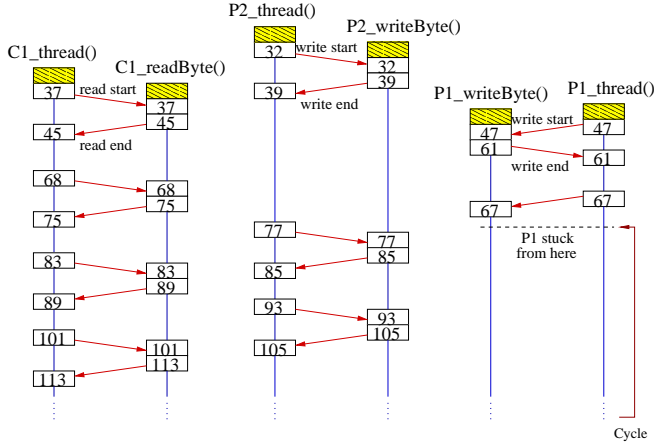


Figure 6. Verification error trace produced by SPIN. The numbers indicate the verification steps and arrows indicate communications between processes through channels.

medium, then whenever a producer initiates a write, it will eventually complete the write”.

In LTL, the property can be expressed as:

$$\begin{aligned} & \Box \Diamond (C_1_read \vee \dots \vee C_n_read) \rightarrow \\ & \Box (P_x_start \rightarrow \Diamond P_x_end) \end{aligned} \quad (3)$$

However, for the case where $m = 2$ and $n = 1$, SPIN reports that the property does not hold. From the error trace (see Figure 6), we can see that there is the possibility of starvation. It is possible for P_2 to keep accessing the medium, which prevents P_1 from ever being able to write, and vice versa.

Table 2. Summary of verification of property 2

Depth reached	589790
States generated	1.65642e+06
State transitions	6.34532e+06
Memory used	317.251 MB
	(Partial Order Reduction)
CPU time elapsed	43.79s

4.2 Constraints and Schedulers

In MMM specification, LTL or LOC constraints can be used to limit the possible behavior of a design. However, downstream synthesis procedure must guarantee that the

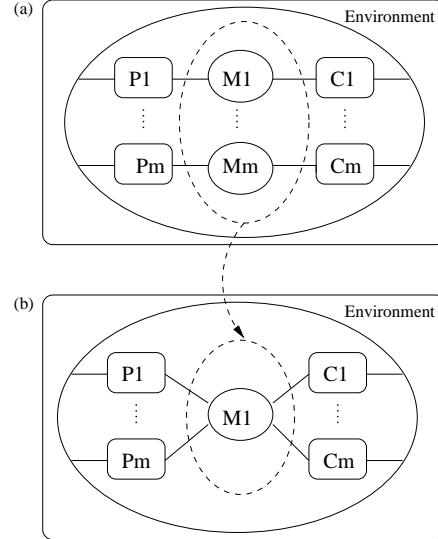


Figure 7. Example of a refinement.

constraints are correctly implemented. If we want the property 3, with $m=2$, $n=1$, and $x=1$, to hold, we may put the following constraint into MMM design:

$$P_1_start \wedge P_2_start \rightarrow \neg P_2_end \text{ U } P_1_end \quad (4)$$

The constraint is “translated” into SPIN environment as the left-hand-side of an implication. Property 3 should be proven only for the cases where the constraint is satisfied. In other word, we prove the property:

$$\text{Constraint}(4) \rightarrow \text{Property}(3) \quad (5)$$

Property 5 is proven by SPIN within one minute of CPU time.

On the lower level of abstraction, constraint 4 can be implemented as a scheduler that has a static-priority policy with P_1 having higher priority. We use SPIN to prove that property 3 holds in the presence of such a scheduler. In addition, if we assign P_2 to have higher priority, the property fails. Another scheduling policy that can be proven to allow property 3 to hold is the round robin scheduling, where the producers take turns accessing the medium.

4.3 Transformation and Refinement

Of course, system level synthesis procedures may not always be driven by the result of functional verification. For example, communication media may be combined to reduce the cost. MMM can be used to formally represent the design before and after a particular synthesis step. Consider the example in Figure 7. In (a), m single-space media are

Table 3. Summary of verification for formula 9

	Regular	Inlining
Depth reached	13224645	1112111
States generated	7.27328e+07	1.49894e+07
State transitions	9.40072e+08	6.6521e+07
Memory Used (Graph encoding)	464.541 MB	101.626
CPU time	9h:44m:48s	31m:54s

used and producer-consumer data streams are running independently. It is trivial to verify that

$$\square (C_x_start \wedge M_x_empty \wedge \neg P_x_start \rightarrow \neg C_x_end \text{ U } P_x_start) \quad (6)$$

where $x \in [1, m]$.

Now, let's consider (b) where only one single-space medium is used. It is derived from (a) through a structural composition with which the following property (which is derived from property 6) is not guaranteed to hold.

$$\square (C_x_start \wedge M_1_empty \wedge \neg P_x_start \rightarrow \neg C_x_end \text{ U } P_x_start) \quad (7)$$

Indeed, SPIN verifies that the property 7 does not hold. The error trace shows that for the property to hold, a constraint must be added such that streams of data do not mix (i.e. if P_x write, then no consumer can read until C_x read):

$$\square (P_x_write \rightarrow \bigwedge_{y \neq x} (\neg C_y_read \text{ U } C_x_read)) \quad (8)$$

With these constraints, the property may be verified by SPIN using the LTL property:

$$\text{Constraint}(8) \rightarrow \text{Property}(7) \quad (9)$$

We verify the design with $m = 2$, which has 113 lines of MMM source code and 836 lines of Promela code after translation. The verification completes without error. However, due to the increase of complexity, it takes more than 9 hours on the 1.5GHz Athlon machine. If we use inlining translation(see section 2.2), the complexity can be largely reduced and only about 30 minutes are needed to complete the exhaustive verification. Table 3 lists the detailed resource usage of these two verifications. From the table, we can see that the total state storage of the inlining translation is much smaller than the original one, which saves both memory and time usage for the verification. This is because the inlining translation uses much simpler data structures to capture the synchronization between MMM constructs. The improvement in efficiency comes at a cost that debugging is now more difficult.

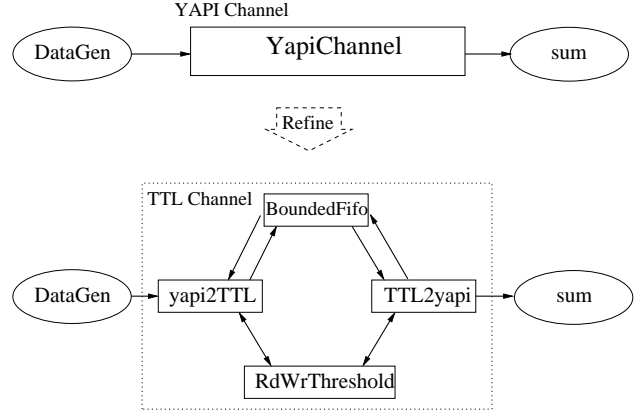


Figure 8. YAPI Channel and TTL Channel.

We have also run a verification session with a dynamic scheduler of the following form: “if P_x writes, then no consumer can consume until C_x does”. As expected, the property pass with similar complexity measurements. Through experimentation, we have found that no round-robin scheduling nor any static priority real-time scheduler allow the property to pass.

4.4 YAPI and TTL

Y-chart Application Programmer’s Interface (YAPI) is a popular model of computation for designing signal processing systems. It is basically a Kahn process network extended with the ability to non-deterministically select an input port to consume and an output port to produce [13]. Within Metropolis, a library environment is set up such that any YAPI design can be written using constructs in the Metropolis library. Central to YAPI MoC is the definition of communication channel and its refinement into Task Transition Level (TTL) [7]. Figure 8 shows how a YAPI channel is refined to a TTL channel. A YAPI channel models an unbounded First-In-First-Out (FIFO) buffer, similar to Kahn process network. Asynchronously, writer process writes the data into one end of the channel and reader process reads the data from the other end of the channel. At the lower level (TTL), the channel is modeled by a bounded FIFO buffer. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. As Figure 8 shows, the TTL channel has a bounded FIFO(*BoundedFifo*) whose size can be set at design time, and a control medium(*RdWrThreshold*) which enforces a protocol to guarantee correctly writing to and reading from the FIFO buffer. To test the YAPI channel and its TTL refinement, we use a writer process(*DataGen*) to write a series of data into the channel and a reader process(*sum*) to read the data from it. One property we want to check is

that there should be no deadlock situation within the channel, i.e. once the writer starts writing data into the channel, it will finish writing eventually. This property can be expressed as an LTL formula:

$$\square (datagen_start \rightarrow (\diamond datagen_finish)) \quad (10)$$

This property is verified on the YAPI level with exhaustive verification in 9 hours with the inlining translation. The relevant verification parameters are listed in Table 4.

Table 4. Summary of verification for YAPI channel

MMM source code	199 lines
Promela code	326 lines
Depth reached	19195
States generated	4.48827e+07
State transitions	6.97818e+07
Memory used	507.469 MB (Partial Order Reduction)
CPU time elapsed	8h:41m:3s

We then proceed to verify the TTL channel as it is exactly described in [13]. Unexpectedly, the deadlock-free property doesn't hold. In less than 1 minutes of CPU time, the verification fails. After the analysis of the error trace (which is similar to Figure 6), a bug is located in the protocol part of the channel (*RdWrThreshold*) in [13]. After correction (adding one statement to allow the writer to be awoken explicitly), we re-run the verification with the existing setup (1.5GHz Atholon and 1GB memory). The verification could not be finished even after it ran for 40 hours and used up the 1GB memory. The refinement process has caused the channel to become much more complex (protocol control is added). Therefore, we use the approximate verification, i.e. the bitstate technique [10], to verify the property on the TTL channel and receive the approximate coverage of at least 98%. All relevant verification parameters are listed in Table 5.

SPIN model checker also provides its own particular mechanism to detect deadlocks. In Promela, designers can explicitly set expected *end* states and let SPIN search the unreachable *end* states that are caused by deadlocks. In our study, we detect deadlocks using an LTL formula, which is functionally equivalent to the *end* state search, but works on the system source code (MMM) rather than Promela. Our deadlock case study is consistent with the idea that the designers should work at the system level specification language (MMM) as much as possible so as not to entangle themselves in detailed implementations or the details of the

Table 5. Summary of approximate verification for TTL channel

MMM source code	720 lines
Promela code	2158 lines
Depth reached	43149
States generated	1.00611e+08
State transitions	1.55842e+08
Total memory used	728.472 MB (Partial Order Reduction)
Approx. coverage	$\geq 98\%$
CPU time elapsed	3h:30m:50s

the verification model (i.e. Promela). An automatic translator from MMM to Promela, as well as the back annotation procedures are used to bridge the gap between the specification model used by the designers and the verification model that is preferred by the developers of the general formal verification tools.

5 Conclusion and Future Directions

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

We are currently working on integrating more system level constructs into the translator, including mapping of functional specification to architectural services. We are also working on the formal verification of platform architectures, where the properties of a platform, as oppose to the properties of the design, will be verified. This will enable the formal understanding, analysis, and verification of the system design from high-level specification all the way down to low-level implementation.

References

- [1] <http://netlib.bell-labs.com/netlib/spin/whatispin.html>, 2003.
- [2] <http://www.cadence.com/products/formalcheck.html>, 2003.
- [3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [4] 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.

- [5] 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.
- [6] X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Formal verification of embedded system designs at multiple levels of abstraction. *International Workshop on High Level Design Validation and Test - HLDVT02*, Sept. 2002.
- [7] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. *Proceedings of International Symposium on System Synthesis*, Oct. 2001.
- [8] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. *Proc. IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification*, June 1993.
- [9] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.
- [10] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in Systems Design*, 13(3):289–307, Nov. 1998.
- [11] G. J. Holzmann and D. Peled. An improvement in formal verification. *Proceedings of FORTE 1994 Conference, Bern, Switzerland*, 1994.
- [12] 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*, 19(12):1523–1543, Dec. 2000.
- [13] E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzter, P. Lieverse, and K. Vissers. Yapi: application modeling for signal processing systems. *Proceedings of the 37th Design Automation Conference*, 2000.
- [14] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Specification. *Springer-Verlag*, 1992.
- [15] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [16] P. Wolper and D. Leroy. Reliable hashing without collision detection. *Proceedings of 5th International Conference on Computer Aided Verification*, pages 59–70, 1993.