Formal Deadlock Analysis of SpecC Models Using Satisfiability Modulo Theories

Che-Wei Chang and Rainer Dömer

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2625, USA

Abstract. For a system-on-chip design which may be composed of multiple processing elements running in parallel, improper execution order and communication assignment may lead to problematic consequences, and one of the consequences could be deadlock. In this paper, we propose an approach to abstracting SpecC-based system models for formal analysis using satisfiability modulo theories (SMT). Based on the language execution semantics, our approach abstracts the timing relations between the time intervals of the behaviors in the design. We then use a SMT solver to check if there are any conflicts among those timing relations. If a conflict is detected, our tool will read the unsatisfiable model generated by the SMT solver and report the cause of the conflict to the user. We demonstrate our approach on a JPEG encoder design model.

1 Introduction

An embedded system design can be implemented in many ways, and a typical design usually consists of hardware and software components running on one or multiple processing elements. In such a design, the partitioned components on different processing elements are executed in parallel. To make sure the data dependency and the execution order is correct, communication between components synchronizes the execution of components on different processing elements. In system-level description languages (SLDLs), like SpecC and SystemC, the communication between components is implemented as channels, and multiple types of channel are provided in the SLDLs to satisfy different kinds of communication and synchronization requirements.

Channels provide a convenient way to communicate among multiple processing elements. However, misusing the type of channel or setting incorrect buffer size in a channel can lead to deadlock situations, and it is difficult to determine the cause of deadlocks when the design is complex. In this paper we propose a method to perform static analysis and detect deadlocks in the design automatically. Based on the SpecC execution semantics, our approach can extract the timing relations between behaviors in the design, and then analyse these with Satisfiability Modulo Theories (SMT) to detect any conflicts. To accelerate the debugging process, our approach also reports the causes of the deadlock to the user if a deadlock situation is found.

G. Schirner et al. (Eds.): IESS 2013, IFIP AICT 403, pp. 116–127, 2013.

This report is organized as follows: in Section 2 we list the related works in formal validation of SLDLs. In Section 3, we briefly introduce SpecC SLDL and Satisfiability Modulo Theories. In Section 4, we describe our proposed approach in detail, including the assumptions and limitations at this point. Also, we illustrate the conversion from SpecC model to SMT assertions. In the last two sections, we demonstrate our approach with a JPEG encoder model and sum up with a conclusion and future work.

2 Related Work

A lot of research has been conducted in the area of verification and validation of system-level designs. We can see that many researchers convert the semantics or behavioral model of the SLDL into another well-defined representation and make use of existing tools to validate the extracted properties. In [3] and [4], a method to generate a state machine from SystemC and using existing tools for test case generation is proposed; in [5] and [6], a SystemC design is mapped into semantics of UPPAAL time automata and the resulting model can be checked by using UPPAAL model checker; [7] proposed an approach to translate SystemC models into a Petri-net based representation for embedded systems(PRES+) which can then be used for model checking. In [8], a SystemC design is represented in the form of predictive synchronization dependency graph (PSDG) and extended Petri Net, and an approach combining simulation and static analysis to detect deadlocks is proposed. [9] focuses on translating a SystemC design into a state transition system, i.e. the Kripke structure, so that existing symbolic model checking techniques can be applied on SystemC.

3 Preliminaries

3.1 SpecC SLDL

SpecC [1] is a SLDL and is defined as extension of the ANSI-C programming language. It is a formal notation intended for the specification and design of digital embedded systems, including hardware and software portions. SpecC supports concepts essential for embedded systems design, including behavioral and structural hierarchy, concurrency, communication, synchronization, state transitions, exception handling, and timing. The execution semantics of the SpecC language are defined by use of a time interval formalism [2].

3.2 Satisfiability Modulo Theories

Satisfiability (SAT) is the problem of determining if an assignment of the Boolean variables exists, that makes the outcome of a given Boolean formula true. Satisfiability Modulo Theories (SMT) checks whether a given logic formula is satisfiable over one or more theories. Unlike the formulas in Boolean SAT which are built from Boolean variables and composed using logical operations, the satisfiability of formula like $(x + 2y \leq 7) \wedge (2x - y \leq 10)$ can be solved by combining a SAT solver with a theory solver for linear arithmetic. The problem to be solved by SMT can be described with richer language (arithmetic and inequality in the example above), and the meaning of the formula will be captured by the supporting theories.

Our proposed approach abstracts the timing relations among the behavior and channel activities in the SpecC model, and then describes the relations in the form of inequality expressed in SMT-LIB2 language. After the formulas are generated, a SMT solver is then used to solve the formula and check if it is satisfiable. In our implementation, we use Z3 theorem prover developed at Microsoft Research as our SMT solver. For more detailed information about SMT-LIB2 language and Z3 theorem prover, please refer to [11] and [10].

4 From SpecC to SMT Assertions

In this section, we first introduce the supported SpecC execution types in our approach and their execution semantics.

4.1 Execution

The basic structure of a SpecC behavior includes port declaration, a *main* method, local variable and function declaration (optional), and sub-behavior instantiation (optional). The supported SpecC behavior models in our tool can be categorized into the following two types:

Leaf Behavior: A behavior is called leaf-behavior if it is purely composed of local variable(s), local function(s) and a *main* method, and there is no sub-behavior instantiation in the behavior. In the example shown in *Figure* 4, behavior *ReadPic* and *Block*1 are leaf behaviors.

Non-Leaf Behavior: A behavior is called non-leaf behavior if it is purely composed of sub-behavior instance(s) and a main method. For non-leaf behaviors, all statements in the *main* method are limited to statements specifying the execution type of the behavior and function calls to sub-behavior instances. In the example shown in *Figure* 4, behavior *DUT*, *Read*, *JPEG_encoder* and *Pic*2*Blk* are non-leaf behaviors.

Note that for simplicity our tool does not support models which do not fit into these two categories, and the execution types we are going to describe in this section is for non-leaf behaviors only since sub-behavior instantiation can only occur in non-leaf behavior.

In SpecC, the sub-behavior or sub-channel instantiation is regarded as a statement of function call to a method of the sub-behavior or sub-channel. To specify the execution time of a statement, for each statement s in a SpecC program, a time interval $\langle T_{start}(s),\ T_{end}(s)\rangle$ is defined. $T_{start}(s)$ and $T_{end}(s)$ represent the start and end times of the statement execution respectively, and the following condition must hold: $T_{start}(s) < T_{end}(s)$

The execution time of an instantiated behavior s $T_{exe}(s)$ is defined as $T_{exe}(s)$ = $T_{end}(s)$ - $T_{start}(s)$. For a statement S consisting of a set of sub-behavior instances $\langle s_{sub\_1}, s_{sub\_2}, s_{sub\_3}, ...s_{sub\_n}\rangle$, the following condition holds:

$$\begin{aligned} \forall i \in \{1,2,3,...n\}, T_{start}(S) &\leq T_{start}(s_{sub\_i}) \\ T_{end}(S) &\geq T_{end}(s_{sub\_i}) \end{aligned}$$

The type of execution defines the relation between the T_{start} and T_{end} of the behavior instance under the current behavior, and it is specified in the *main* method of the behavior. In the following, four types of supported execution are described, which are *Sequential*, *Parallel*, *Pipelined*, and *Loop* execution. *Figure* 1 shows an example of specifying *Sequential*, *Parallel* and *Pipelined* execution in a SpecC behavior. *Loop* execution is not a explicitly defined behavioral execution in SpecC, but we can regard it as a special case of *Pipelined* execution with only one instance inside.

```
behavior B_seq
{
  B  b1, b2, b3;

  void main(void)
  {
    b1.main();
    b2.main();
    b3.main();
  }
};
```

```
behavior B_par
{
  B_seq  A, B;

  void main(void)
  {  par {
       A.main();
       B.main();
     }
  }
};
```

```
behavior B_pipe
{
 B  b1, b2, b3;

 void main(void)
 { pipe(i=0; i<N; i++){
     b1.main();
     b2.main();
     b3.main(); }
 }
};
```

```
behavior B_loop
{
 B  b1;

 void main(void)
 { pipe(i=0; i<N; i++){
     b1.main(); }
 }
};
```

Fig. 1. Four Supported Execution Types

Sequential Execution. *Sequential* execution of statements is defined by ordered time intervals that do not overlap. Formally, for a statement S consisting of a sequence of sub-statements $\langle s_1, s_2, ...s_n\rangle$, the time interval of statement S includes all time intervals of the sub-statements, and the following conditions hold:

$$\begin{aligned} \forall i \in \{1,2,...,n\}, T_{start}(S) &\leq T_{start}(s_i) \\ T_{start}(s_i) &< T_{end}(s_i) \\ T_{end}(s_i) &\leq T_{end}(S) \\ \forall i \in \{1,2,...,n-1\}, T_{end}(s_i) &\leq T_{start}(s_{i+1}) \end{aligned}$$

Note that sequential statements are not necessarily executed continuously. Gaps may exist between T_{end} and T_{start} of two consecutive statements, as well as between the T_{start} (T_{end}) of the sub-statement and the T_{start} (T_{end}) of the statement in which the sub-statement is called. *Figure* 2 shows an example of the time interval for the sequential execution in *Figure* 1.

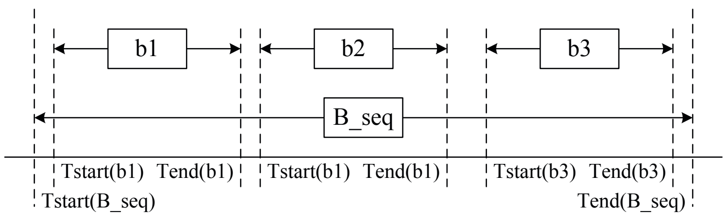

Fig. 2. Time interval for sequential execution

Parallel Execution. *Parallel* execution of statements can be specified by *par* or *pipe* statements. In particular, the time intervals of the sub-statements invoked by a *par* statement are the same. Formally, for a statement S consisting of concurrent sub-statements $\langle s_1, s_2, ...s_n\rangle$, the following conditions hold:

$$\begin{aligned}\forall i \in \{1,2,...,n\}, T_{start}(S) &= T_{start}(s_i)\\ T_{end}(S) &= T_{end}(s_i)\\ T_{start}(s_i) &< T_{end}(s_i)\end{aligned}$$

Figure 3 shows an example of the time interval for the parallel execution in *Figure* 1.

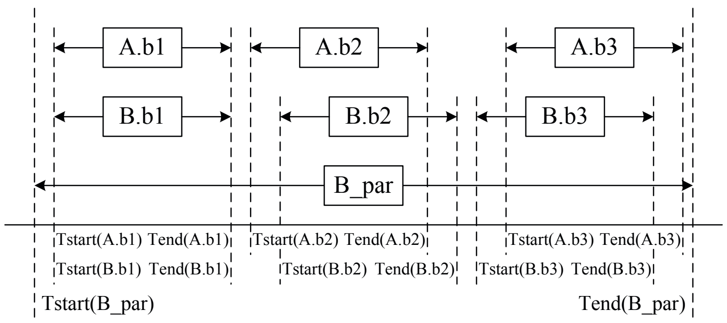

Fig. 3. Time interval for parallel execution

Pipelined and Loop Execution. *Pipelined* execution of statements is a special form of concurrent execution. The syntax of *pipe* statement in SpecC is illustrated in *Figure* 1, where N in the example specifies the number of iterations. Formally, for a statement S consisting of sub-statements $\langle s_1, s_2, ...s_n\rangle$ executed for m iterations in pipelined manner, let $s_{i \cdot j}$ represents the j-th iteration of the execution of statement s_i. Then the following conditions hold:

$$\begin{aligned}
&\forall i, x \in \{1, 2, ..., n\},\ j, y \in \{1, 2, ..., m\} :\\
&T_{start}(s_{i \cdot j}) < T_{end}(s_{i \cdot j}),\\
&T_{start}(s_{i \cdot j}) = T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y\\
&T_{end}(s_{i \cdot j}) = T_{end}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j = x + y\\
&T_{end}(s_{i \cdot j}) \leq T_{start}(s_{x \cdot y}),\ \ \textit{if}\ \ i + j < x + y
\end{aligned}$$

Loop execution is not defined explicitly in the behavioral execution semantics of SpecC, but it can be regarded as a special case of *Pipelined* execution with only one sub-statement.

Note that in the definition of *pipelined* statements the iteration number could be infinity if the number is not specified, i.e. no range specification after the statement *pipe*. However, to simplify the static analysis in this proposed method, at this point, the number of iterations has to be a finite integer and explicitly specified in the model.

Please be aware that for now our proposed method does not support all types of execution and communication defined in SpecC. Full support of SpecC is part of our future work.

4.2 Communication

In SpecC, the communication between two behaviors can be implemented by port variable, channel communication, or by accessing global variables. Since right now the goal of our approach is to detect deadlocks in the design, the communication implemented with port variables and global variables are not taken into consideration because they will not lead to deadlock situation in the design.

Multiple types of channels are defined in SpecC. These include semaphore, mutex, barrier, token, queue, handshake, and double handshake. In this paper, we use queue channel with different buffer sizes to model the supported channel communication in our approach. For example, to model the blocking characteristics of handshake channels, we use a queue channel with one element buffer and zero element buffer to implement the handshake and double handshake channel.

To clearly identify the communication between behaviors, we also impose some limitations on the communication between behaviors. First, to make the data dependency between behaviors clear, we limit the communication between behaviors to point-to-point, i.e. every instantiated channel in the design is dedicated to the communication between a pair of sender and receiver. Second, to abstract the channel activity without looking into too much detail of the behavior model, the function call of the sending (receiving) function to (from) a certain channel can only be executed once in the *main* method of a behavior, i.e. a function call to channel communication in any type of iteration (for or while loop) in the *main* method of a behavior model is not supported. For the case that the output of a behavior has to be separated into multiple parts and sent to another behavior, the sending (receiving) function calls have to be wrapped in a behavior and executed in loop execution by using *pipe* statements.

Figure 4 shows an example of the situation described above. In this example, a small picture of size 24-by-16 pixels is read and encoded into a JPEG file. Since the input image block size for a JPEG encoding process is eight-by-eight pixels, the picture has to be separated into 6 sub-blocks. The raw picture is read into the topmost behavior *DUT* by sub-behavior *Read*, then behavior *Pic2Blk* divides the picture into six 8-by-8-pixel blocks and sends the blocks to JPEG encoder model. Inside behavior *Pic2Blk*, behavior *Block*1 is instantiated in a loop execution. Behavor *Block*1 fetches the block from the raw picture according to the current iteration number, and calls the sending function to send the data to JPEG encoder through channel Q. In this example, channel Q is a queue channel with two buffers and each buffer is an integer array of size 64.

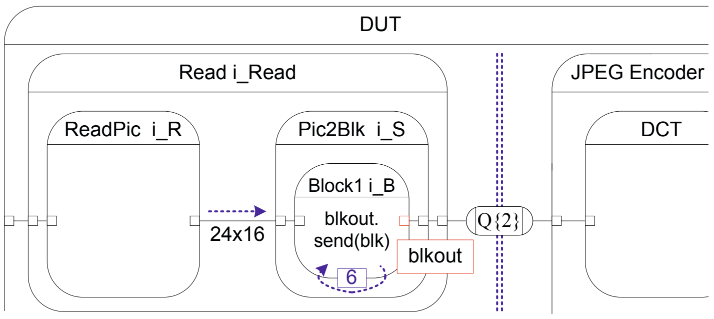

Fig. 4. Behavior *Read* in the JPEG encoder SpecC model

Similar to the time interval $\langle T_{start}, T_{end}\rangle$ defined for the execution of a statement, a time stamp set $\langle T_{sent}(Q), T_{rcvd}(Q)\rangle$ is also defined for each channel communication activity between behaviors, where T_{sent} represents the time stamp when the the execution of sending a data to the channel finishes, and T_{rcvd} represents the time stamp when the execution of receiving data from the channel finishes. Based on the definition of T_{sent} and T_{rcvd}, for a queue channel Q communication through which m data items are transferred, the relation between time stamps $T_{sent}(Q_i)$ and $T_{rcvd}(Q_i)$, where Q_i represents the i-th data transfer through channel Q, should hold:

$$\begin{aligned}\forall i \in \{0,1,2,...,m-1\}, T_{sent}(Q_i) &\leq T_{rcvd}(Q_i)\\ \forall i \in \{0,1,2,...,m-2\}, T_{sent}(Q_i) &< T_{sent}(Q_{i+1})\\ T_{rcvd}(Q_i) &< T_{rcvd}(Q_{i+1})\\ \forall i \in \{0,1,2,...,m-n-1\}, T_{rcvd}(Q_i) &\leq T_{sent}(Q_{i+n})\end{aligned}$$

where n is the buffer depth of channel Q.

4.3 From Time Stamps to SMT Assertions

Figure 5 shows the flow of our proposed method. First, the SpecC model is converted into a design representation called SpecC internal representation(SIR).

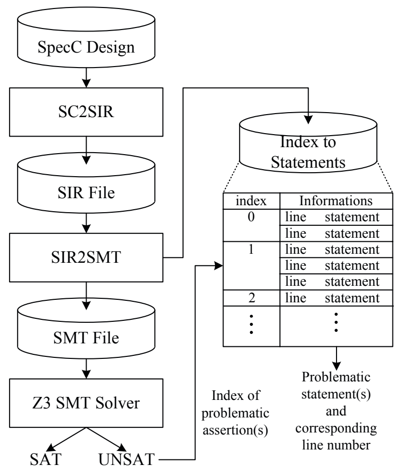

Fig. 5. The flow of converting a SpecC model into SMT assertions and deadlock analysis with the Z3 SMT solver

The next step is to traverse the internal representation structure and generate the assertions corresponding to the statements in the design. At the same time, an index-to-statement record is created which links the generated assertions to the statements in the design. After the assertions and records are generated, we use the Z3 theorem prover to check if there is any conflict in the set that makes the equations unsatisfiable. If there are any, Z3 will report the indices of assertions leading to the conflict, and our tool can use the indices to access the record and report the problem information to the user. In the following part of this section, we use the model shown in *Figure* 4 as an example, and illustrate the corresponding assertions for the model.

Execution to SMT Assertions: In our proposed method, we use uninterpreted functions in SMT-LIB2 language to represent every time stamp in the model, and convert the timing relations between those stamps into assertions. For an uninterpreted function, the user can define the number of arguments, the data type of argument, the data type of the return value, and its interpretation. In our method, the return value of an uninterpreted function is seen as the value of a time stamp, and the argument(s) of the function is (are) used to specify the number of times a behavior instance is executed in a pipelined structure or a loop. For a behavior instance, which is not in a pipelined or loop execution, the time stamps of this instance are represented as uninterpreted functions with no argument since the behavior will only be executed once.

For example, for instance *i_S* in behavior *Read* in *Figure* 4, the following assertions will be generated:

$$
\begin{aligned}
&(declare-fun\ \ T_{start}DUT.i\_Read.i\_S\ \ ()\ \ Int)\\
&(declare-fun\ \ T_{end}DUT.i\_Read.i\_S\ \ ()\ \ Int)\\
&(assert\ (<=\ T_{start}DUT.i\_Read\ \ T_{start}DUT.i\_Read.i\_S))\\
&(assert\ (<=\ T_{end}DUT.i\_Read.i\_S\ \ T_{end}DUT.i\_Read))\\
&(assert\ (<\ T_{start}DUT.i\_Read.i\_S\ \ T_{end}DUT.i\_Read.i\_S))\\
&(assert\ (<=\ T_{end}DUT.i\_Read.i\_R\ \ T_{start}DUT.i\_Read.i\_S))
\end{aligned}
$$

For a behavior instance, which is executed in a pipelined or loop for multiple times, the time stamps of this instance are represented as uninterpreted functions with one or multiple arguments. The input value of the argument is the number of execution times of this instance.

For example, for instance $S1$ in behavior $Sender$ in $Figure$ 4, the following assertions will be generated:

$$
\begin{aligned}
&(declare-fun\ \ T_{start}DUT.i\_Read.i\_S.i\_B\ \ (Int)\ \ Int)\\
&(declare-fun\ \ T_{end}DUT.i\_Reae.i\_S.i\_B\ \ (Int)\ \ Int)\\
&(assert\ \ (forall\ \ ((I0\ \ Int))\ \ (=>\ \ (and\ \ (>=\ \ I0\ \ 0)\ \ (<=\ \ I0\ \ 5))\\
&\quad(<=\ \ T_{start}DUT.i\_Read.i\_S\\
&\qquad(T_{start}DUT.i\_Read.i\_S.i\_B\ \ I0)))))\\
&(assert\ \ (forall\ \ ((I0\ \ Int))\ \ (=>\ \ (and\ \ (>=\ \ I0\ \ 0)\ \ (<=\ \ I0\ \ 4))\\
&\quad(<=\ \ (T_{end}DUT.i\_Read.i\_S.i\_B\ \ I0)\\
&\qquad(T_{start}DUT.i\_Read.i\_S.i\_B\ \ (+\ I0\ \ 1))))))
\end{aligned}
$$

Communication to SMT Assertions: In our approach, the time stamp of every channel activity is represented as an uninterpreted function with one argument, and the input value of the argument is the number of execution times of channel activity. For example, for channel Q in behavior DUT in $Figure$ 4, the following assertions will be generated:

$$
\begin{gathered}
\forall i \in \{0,1,...,5\},\ \ T_{sent}DUT.Q(i) \leq T_{rcvd}DUT.Q(i)\\
\forall i \in \{0,1,...,3\},\ \ T_{rcvd}DUT.Q(i) \leq T_{sent}DUT.Q(i+2)
\end{gathered}
$$

Our tool will also generate the equality for the time stamp of the channel activity and the time stamp of the function call to the interface of the corresponding channel. For example, the following assertion will be generated for the channel accessing function call $blkout$ in $Figure$ 4:

$$
\begin{aligned}
&\forall i \in \{0,1,...,5\},\\
&T_{sent}DUT.q(i) = T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i)\\
&T_{start}DUT.i\_Read.i\_S.i\_B(i) \leq T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i)\\
&T_{sent}DUT.i\_Read.i\_S.i\_B.blkout(i) \leq T_{end}DUT.i\_Read.i\_S.i\_B(i)
\end{aligned}
$$

For space limitations, we can only list a portion of the assertions as examples. Other assertions are generated based on the timing relations we described in *Section* 4.1 and *Section* 4.2.

During the assertion creation, a table named *index-to-statement* will also be generated. For every assertion generated by our tool, an identical index is given to the assertion and the information about the corresponding statement that is stored in the entry addressed by that index. Take assertion $T_{rcvd}DUT.Q(i) \leq T_{sent}DUT.Q(i+2)$ listed above as an example. This assertion is generated because channel Q is instantiated in behavior DUT and its depth is set to two. Therefore, in the entry addressed by the index of this assertion, the information of the statement specifying the depth of the channel is stored.

5 Experiments

In this section, we demonstrate our proposed method with a JPEG encoder SpecC model. In this example, the JPEG encoder is asked to encode five subframes of size eight-by-eight pixels from a raw picture. *Figure* 6 shows two different implementations of the SpecC JPEG encoder model.

In the JPEG encoder, every subframe will be encoded in three steps, two-dimensional discrete cosine transform (DCT), quantization, and Huffman encoding. For every subframe, these three encoding steps have to be executed in order. In our SpecC model, three behaviors D, Q, and H are implemented to perform the discrete cosine transform, quantization, and Huffman coding of JPEG encoding, respectively.

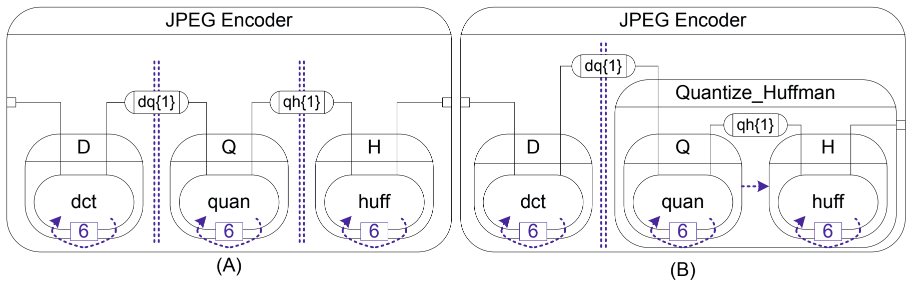

Fig. 6. Two examples of JPEG encoder SpecC model. Example(A) is a design without deadlock, while Example(B) will incur a deadlock situation.

As shown in *Figure* 6(A), behavior D, Q, and H are executed in parallel fashion. To make sure these three steps are executed in correct order, two queue channels are used to transfer the intermediate encoding data between these three behaviors, instead of using port variable connections. In model (B), sub-behavior Q and H are wrapped into a behavior $Quantize\_Huff$ and executed in sequential manner. The problem in model (B) is that behavior Q will halt forever after the first two iterations of its sub-behavior *quan*. In this composition, behavior *quan* will be executed six times before the execution of behavior Q finishes, but the execution will stop because the queue channel between behavior *quan* and behavior $huff$ becomes full after the first two data sets are generated. Since behavior H can only be executed after the execution of behavior Q finishes, the sub-behavior $huff$ cannot be executed to empty the queue channel qh.

We have used our tool to analyse both models. $Table$ 1 shows the analysis results of the two models.

Table 1. Static SMT analysis results for model (A) and (B)

$Design$	$\#ofAssertions$	$Time$	$Satisfiability$	Error Report
Model-(A)	187	4.94s	SAT	N/A
Model-(B)	192	1.39s	UNSAT	Type: QUEUE Line[16]: Channel[qh] Type: SEQ Line[23]: Instance[Q] Line[24]: Instance[H] Type: LOOP Line[58]: Behavior[Q] Line[60]: Instance[quan] ...

In $Table$ 1, the value in $Line$ represents the line number of the statement in the SpecC model, and $Type$ shows the type of information stored in the entry. For example, the $Type : SEQ$ in this table shows that behavior instance Q and H are executed in sequential manner, and Q is executed before H. Though for now the error report might not be intuitive for the unfamiliar user to understand what led to the deadlock, the model designer who developed the model will easily recognize the deadlock situation.

6 Conclusion

In this paper we have proposed an approach to statically analyze deadlocks in SpecC models using a SMT solver. After the introduction of four supported execution types and queue channel communication in our tool, we have described our approach in detail by showing how to extract timing relations between time stamps according to SpecC execution semantics, and have illustrated the conversion from timing relations to SMT-LIB2 assertions. Finally we demonstrated our implementation with a JPEG encoder model, and showed that our approach is capable of detecting the deadlock in the model and reporting useful diagnostic information to the user.

At this point, this research is still far from complete and there is a lot of future work to do. Future work includes expanding the support for larger models, and extending our research to cover more design verification problems. For now our implementation only supports a confined set of SpecC models and leaves some important features of SpecC unsupported, such as FSM composition. In the future we will improve our tool so that it can extract the timing relations from a control-flow graph and represent the relations with SMT-LIB2 assertions. Except for the deadlock analysis, we also found that SMT solver might be suitable for time constraint analysis. We will keep exploring possible applications and make this approach more general in the future.

Acknowledgment. This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523. The authors thank the NSF for the valuable support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Dömer, R., Gerstlauer, A., Gajski, D.: SpecC Language Reference Manual Version 2.0, `http://www.cecs.uci.edu/~specc/reference/SpecC_LRM_20.pdf`
2. Fujita, M., Nakamura, H.: The Standard SpecC Language. In: Proceedings of the International Symposium on System Synthesis, Montreal (October 2001)
3. Habibi, A., Moinudeen, H., Tahar, S.: Generating Finite State Machines from SystemC. In: Design, Automation and Test in Europe, pp. 76–81 (2006)
4. Habibi, A., Tahar, S.: An Approach for the Verification of SystemC Designs Using AsmL. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 69–83. Springer, Heidelberg (2005)
5. Herber, P., Fellmuth, J., Glesner, S.: Model Checking SystemC Designs Using Timed Automata. In: Int. Conf. on HW/SW Codesign and System Synthesis. ACM Press (2008)
6. Herber, P., Pockrandt, M., Glesner, S.: Transforming SystemC Transaction Level Models into UPPAAL timed automata. In: 2011 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 161–170 (2011)
7. Karlsson, D., Eles, P., Peng, Z.: Formal verification of SystemC Designs using a Petri-Net based Representation. In: DATE, pp. 1228–1233 (2006)
8. Chou, C.-N., Ho, Y.-S., Huan, C.H.C.-Y.: Formal Deadlock Checking on High-Level SystemC Designs. In: 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 794–799 (2010)
9. Chou, C.-N., Ho, Y.-S., Huan, C.H.C.-Y.: Symbolic Model Checking on SystemC Design. In: Proceedings of the 49th Annual Design Automation Conference, DAC 2012, pp. 327–333 (2012)
10. Z3 theorem prover, `http://z3.codeplex.com/`
11. Cok, D.R.: The SMT-LIB v2 Language and Tools: A Tutorial, `http://www.grammatech.com/resources/smt/SMTLIBTutorial.pdf`