**Specification of Exception Handling in Grammar-based Hardware Synthesis**

Johnny Öberg¹, Anshul Kumar², Ahmed Hemani¹

¹Electronic Systems Design Laboratory, Royal Institute of Technology, ESDLab/KTH-Electrum, Electrum 229, S-164 40 Kista, Sweden
Telephone: + 46 8 752 13 05
Email: {johnny, ahmed}@ele.kth.se

²Dept. of Computer Science & Engineering, Indian Institute of Technology, New Delhi, India.
Telephone: +91 11 666979 ext. 6002
Email: anshul@cse.iitd.ernet.in

**Abstract**

ProGram is our in-house grammar based specification language targeted to specification of protocols, interfaces and control dominated functionality. The ProGram compiler implements such specifications in hardware. Specification of exception handling functionality and its automatic implementation are key requirements for a robust design methodology. In this paper, we present extensions to the ProGram language for specification of a wide range of exception handling functionality: reset to deal with exception handling needs of global nature, interrupt for representing exception handling on a hierarchical basis and error sequences to handle situations when inputs are not according to the specified grammar.

1. Introduction

Exception handling is necessary to be able to describe hardware and real-time systems in an efficient way and is a key aspect of any system functionality, where it takes the form of reset sequences, interrupt service routines and error handling. The key characteristic of exception handling is its global nature. For instance, a reset would bring the state machine to a known state from any state and clear the storage elements. Reset, though a common hardware construct, does not have any explicit support in VHDL, in which any exception of global nature is treated as any other condition. This makes it very cumbersome to specify exception handling functionality. Other languages, like SpecChart[1][2], have explicit support for specification of exception handling functionality.

Interrupts are of a more local nature than a reset, and specifies how the protocol should react if a high-priority situation occurs, caused by some hardware failure or some illegal operation performed elsewhere in the system, and the parsing of the current input must be terminated immediately.

Error handling is crucial since it makes it possible to detect and recover from error situations during run-time execution of the protocol.

Previously, in [10][11][12], we have presented grammar based specification and synthesis as an effective tool for design of interfaces and protocols. Recognising that exception handling is a fundamental requirement of our application domain, we have extended our previous work to be able to specify exception handling functionality and methods for its automatic implementation. In this paper we introduce the concepts of reset, interrupt and error sequence specification in the grammar. We also introduce the condition to be able to handle small exceptions from the main stream. To be able to change input-streams, we have the redirection scheme, first introduced in [10]. Here we extend the concept of redirections, to make the code more readable. In [10], we also introduced the Other-bit construct. It is the simplest cases of exception and works as an else-statement. The Other-bit construct is described here for completeness of the paper.

Section 2.) surveys related work. Section 3.) reviews our previous work. Section 4.) describes the extensions to our language - ProGram (Protocol Grammar) - for specifying exception handling functionality. Finally, section 5.) concludes with a discussion on the new language constructs.

2. Related Work

Bloks [3], presents Protocol Grammar Interface Language (ProGrIL), based on context free LL(1) grammar extended with attributes. ProGrIL is implemented as a push down automaton, that can handle three types of exceptions: the standard event, the reset event and the input event. The standard event interrupts the execution stream after completion of the current instruction. The reset event is used to implement a restart and a clean start-up of the protocol engine. The input event checks for a particular values on an input and is automatically invoked if an the value appears on the input during parsing of a rule where input events on this input is allowed. The ProGrIL approach has two backs: 1) the code is difficult to read as it is based on mathematical notation and 2) the synthesis engine emulates a software compiler by generating
microcode for a predefined grammar processor, resulting in inefficient hardware; this is in contrast to our and Clairvoyant/DALI strategy of generating a custom hardwired solution.

Clairvoyant’s approach based on regular expression [4][5][6][7] has been developed further into a commercial tool DALI [8] which has a graphical interface and some additional constructs like condition which makes it possible to implement exception handling. DALI also have a special conditional construct that terminates the parsing of the current input if the condition is true. This is similar to our interrupt-case, described later in this paper. The main difference between DALI and ProGram is ProGram’s ability to specify input and output sequences independent of the size of the ports. Clairvoyant/DALI facilitates a clock true description, where every input and output sequence needs to be specified according to the port width constraints. ProGram treats the grammar specification as a high-level specification and uses constraints to specify how the final clock-true implementation should be build up.

3. The ProGram grammar Specification

The ProGram language is based on a regular LL(1) grammar and uses a BNF-like notation [9] to code both input and output sequences. The input production rules specifies in an hierarchical way all valid sentences that the protocol should recognise and the output production rules specifies all valid answers to the valid inputs. Incoming and outgoing messages are coded as bit-streams. Each incoming message can have several output messages associated with it.

The input sequences are partitioned into tokens, the size of the input stream and reduced during the synthesis steps to remove non-determinism from the internal representation, the grammar DAG. The output sequences are partitioned into tokens, the size of the output streams, and scheduled over the available input tokens.

The ProGram specification is divided into five sections: Interface Declarations, Terminal Token Definitions, Memory Layout Productions, Action Macro Productions and Protocol Grammar Productions. In the Interface Declaration Section, interfaces to the external world and between internal sub-protocols and memories are specified. The Protocol Grammar Productions section defines the language of the protocol is specified. The language is given as a hierarchical list of production rules. Each production defines a legal sentence or part of a legal sentence.

4. Exception Handling extensions to ProGram

To allow specification of exception handling functionality, two sections of the ProGram description have been modified. In the interface section, a construct for specifying asynchronous or synchronous reset behaviour of internal registers and output ports has been added. The default is synchronous behaviour. In the grammar rule section, two changes have been made. First, to ease specification of complemented behaviour, the ability to specify all bit-combinations except a specific constant token has been added. Since the tokens will be split and joined later during the synthesis step WordAlignTerminals, the negated token is expanded into its complementary set. Second, several constructs for handling common situations during exception handling have been added and some old ones have been modified.

The condition construct has been added to be able to specify synchronisation situations and to terminate loops. The interrupt and reset statement has been added to enable the designer to specify interrupt and reset sequences.

The first version of ProGram [10] had three rudimentary exceptions in the grammar rule section: the redirection construct, the Other- and Any-bits constructs, plus an error construct to support specification of error conditions in the input sequences. The first two have been extended or modified to increase the readability and expressiveness of the grammar description. The error condition has been extended to include error sequences. Error sequences makes it possible to perform error-recovery.

A. Redirection Exception

A redirection replaces the current input streams with another one. The redirection exception is crucial to write compact and readable code. Without a redirection, the input stream of the tokens in the grammar would have to include all inputs and their allowed combinations that are accepted by the grammar. A simple task like searching for a ‘1’ on a consecutive series of input in sequence would become troublesome to specify, see Figure 1.

There are three types of redirections. Redirections of
type one change the bit-stream locally for all alternatives of a production rule. Redirections of type two change the bit-stream, when the production rule is called. This works as a parameterisation, and makes it possible to use production rules in a generic manner, enabling the same set of rules to be parsed from any set of inputs. Redirections of type three change the bit-stream locally on the alternative where it is specified.

B. Condition Exception

A condition is used when the ordinary stream should be parsed, but when the correct branch to select depends locally on an extra input signal condition. Thus, this can be viewed as a local, temporary, redirection, associated with a single token. The condition is useful for synchronisation with other FSMs and for specification of exit conditions in loops, which are specified as recursive rules. Examples of these features are shown in Figure 2.

A condition can be either left associative or right associative. A left associative condition is associated with the first terminal token to the left of the condition. Thus, it can be used to terminate a generic loop, since only the exit token would be visible from the outside. A right associative condition is associated with the first terminal token to the right of the condition. A generic sequence can in this way be blocked from execution until the synchronisation condition is true.

C. Interrupt and Reset Exception

An interrupt sequence specify how the FSM should react if a high-priority condition arises. The interrupt can be caused by some hardware failure or some illegal operation performed elsewhere in the system. When the interrupt occur, the parsing of the current token sequence is immediately terminated. The interrupt is local and hierarchical. Thus, only alternatives specified at the same level and below as the interrupt was specified can be inter-

The reset statement is similar in semantics to the interrupt statement but is more global in its nature: it can only be placed at top level rule, specified by the start rule, and it resets the whole machine. Both the interrupt and the reset continues with the token sequence following the exception and does not return to the place where the exception occurred. Examples of interrupt exceptions are shown in Figure 3.

D. Error Exception

The error statement specifies the response of the FSM to illegal sequences. Any token sequence that is not in the grammar will result in an error exception. Like interrupt and reset statements, control is not returned to the place where error occurred after responding to the error condition. The error sequence is associated to all bit sequences that come at the same depth or later, counted from the start of the input frame. Also, error statements occurring later in the protocol takes precedence over the error statements that comes on a higher level in the protocol hierarchy. This can be used to recover from faulty transmissions and out-of-sync situations, see Figure 4.

In the case when not all possible input sequences have been specified, there is an implicit error that will trap these sequences. The protocol will then end up in the default error state. The default error state then restarts the parsing from the beginning of the input frame.

E. The Other-bits and Any-bits Exception

The other-bits statement allow the designer to specify a catch-all-that-has-not-been-previously-defined alternative to the previously specified alternatives. In a sense, this is similar to the else-construct supported in ordinary programming languages. The other-bits statement thus allows the designer to specify the complement of what has previously been defined in other alternatives. The ability to specify an else-statement is crucial to achieve compact code. Without this possibility, the user would have to speci-
ify all alternatives, which quickly becomes cumbersome when the number of bits in the token sequence is large, see Figure 5.

The *any-bits* statement is similar to the other-bits statement, but instead of specifying the complement of the token sequences, it specifies a don’t care situation on the input stream, i.e. both a ‘0’ and a ‘1’ is an acceptable input. Thus, the transition should be taken regardless of the values on the input stream.

5. Conclusion and Future Work

We have extended our previous methodology with exception handling. Exception handling is necessary to be able to describe real-time systems in an efficient way. The language constructs that we have added to the ProGram language allows for efficient specification of interrupts, reset and error sequences. *Interrupt* sequences allows to handle events local to parts of the protocol. The *reset* sequence allows to handle start-up and initialisation situations that can happen anywhere within the protocol. The *error* sequence allows for the possibility to detect and recover from error situations during run-time execution of the protocol. Several additions to existing types of exceptions have also been done. The *Other-bit* exception allows to specify complementary bit-patterns and take actions if none of the other specified patterns trigger. The *redirection* allows for changing of the bit-stream inside the protocol and the *condition* allows to specify special exception conditions that are local to a single token. The synthesis algorithms have been implemented and tried out and works as expected [13].

A problem with how the semantics of the exception handling constructs in ProGram is that control is not returned to the place where the interrupt occurred. Instead, the protocol resumes the parsing from the start of the input frame. This is more the semantics of a traditional break statement. It might be necessary in the future, to avoid confusion, to change the semantics of the interrupt to the traditional one and add a break statement to implement the current functionality.

References


References