Automated Formal Verification of the Refined Specification of Digital Systems in HSSL

L. Maron and D. Macko
Faculty of Informatics and Information Technologies,
Slovak University of Technology,
Bratislava, Slovakia
lubomir.maron@stuba.sk, dominik.macko@stuba.sk

***Abstract*—Design of modern hardware systems becomes difficult because of the increasing complexity. As a result, more abstraction is used in the design process. However, an error made at a higher abstraction level is transferred to lower levels. It becomes costly to correct such an error at later design stages, and therefore it must be revealed as soon as possible. The specification language HSSL provides techniques that can help to minimize the possibility of human error at the specification stages. HSSL is intended for formal behavioral specification of hardware and software and it enables formal verification of refined specifications. In this paper, a tool is described that is intended for automated formal equivalence checking of specifications at respective refinement stages. The tool is able to apply refinement rules and to prove that the two specifications describe the same system function. The automation of this process unburdens the designer from time-consuming manual effort. It is especially useful for inexperienced designers, in the education process, which would like to quickly verify their refined specifications.**

***Index Terms*—Computer-aided design, design automation, formal specifications, formal verification, verification automation.**

I. Introduction

For students and other beginning designers, it is almost impossible to design a hardware system that is completely without errors [1]. Functional verification by simulation of a design may not reveal some corner-case errors, which may be masked, and their subsequent correction (i.e. debugging) is difficult – it takes money and time.

The development of a non-trivial hardware/software system starts by the specification, which can be formal or informal. Formal specification is a mathematical expression of software or hardware that is used in their development and implementation [2]. Formal expression helps to identify and express requirements of a system under development, and also helps to verify the system functionality. Formal specification can describe different aspects of the system, such as behavior (Petri nets, finite state machines), data structures (abstract data types, object-oriented languages), or system properties (using various logics – e.g. the First order logic).

Verification is a complex process of checking whether the system is correct. Formal verification proves or disproves its correctness based on the formal specification. Formal verification can have many forms that are based on the three basic formal approaches – equivalence checking, model checking, and theorem proving. A form of equivalence checking is also used in [3], where the verification process is carried out over two specifications, one of which is a refinement of the other. The refinement process can be described as adding more details to the original specification (such as a communication protocol with the system environment). The refined parts of the specification describe the system in lower abstraction – it is closer to the actual system. The first stage of the specification contains the most abstract view of the system, and through the refinement process it reveals the details about the system functionality. The usual hardware-system design process is illustrated in Fig. 1. It starts from an ESL model (Electronic System Level), which is refined to an RTL model (Register-Transfer Level) usually described in some HDL (Hardware Description Language), which is then synthesized into a physical model that can be manufactured.

The refinement process at the specification stages (ESL) is very useful in education. The students would just not understand a detailed complex specification of some system. Instead, they must deal with a simple abstract specification, which is easily understandable, and they use the predefined refinement steps to obtain the detailed specification. To prevent a complete redesign of the system in case of an error, they use formal equivalence checking at the respective refinement stages to verify that the system specification is still correct (i.e. equivalent to the original specification). However, the manual equivalence-checking process is quite difficult, it takes time and errors can be made. Its automation would help the students to quickly verify their designs and also to evaluate the results by a teacher.

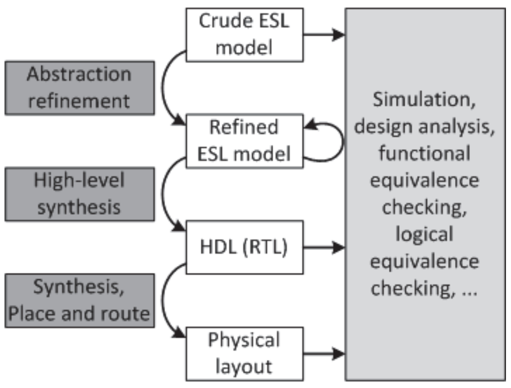

Crude ESL model
Abstraction refinement
Refined ESL model
High-level synthesis
HDL (RTL)
Synthesis, Place and route
Physical layout
Simulation, design analysis, functional equivalence checking, logical equivalence checking, ...

Figure 1. Overview of the usual design flow

Therefore, the goal of this work is to simplify and automate the process of formal verification. It is achieved by the implementation of a tool called HSSL Verification Tool. It is intended to serve mainly for the educational process; however, it can be also used in professional system development.

This paper is organized as follows. In Section II, related work is described, which is focused on HSSL-related design-automation tools and other existing methods of formal-verification automation. A brief introduction to the HSSL refinement process is given in Section III. In Section IV, the development of HSSL Verification Tool is described – the specified requirements, the used methods, and the implemented environment. Before the conclusion, a simple example is provided.

II. Related Work

The main development of Hardware/Software Specification Language (HSSL) [1] has been centered at the Slovak University of Technology in Bratislava. Multiple researchers have continuously participated in HSSL improvement by extending its features or by implementing design-automation tools. For example, the latest HSSL extension is the support of power-management specification for development of power-efficient digital systems [4]. The design-automation tools developed for HSSL include, for example, a functional simulator of HSSL model [5] or a high-level synthesis tool for transformation of the HSSL specification into a VHDL model using Petri nets [6]. Our work supplements the existing tools and provides an alternative to the HSSL-model verification based on time-consuming functional simulation. Especially at the early stages, when the simulation is not possible due to insufficient amount of details.

A simple approach to automate formal verification in VDM++ specification language has been published in [7]. The approach is based on an extension of the language, which enables annotation of relationships between specifications. It then uses an automated translation of such an extended model to a formal language, which has already been supported by a verification system. In our case, we utilize the features the language already has; therefore, our approach minimizes manual overhead.

A different automated approach is used in [8]. It transforms AADL (Architecture Analysis and Design Language) specification to constructs in timed automata, which serve as an input of the existing model checker. This approach is dependent on a third-party tool, and thus only a subset of AADL constructs is verified. Also, in case of [9], the specification is transformed to an algebraic model, on which the existing model checker proves the properties. We avoid transformation to another language/model that is supported by a third-party tool (with its own limitations). We use only the HSSL model, and thus we are constrained only by the support of its constructs in the developed tool.

There is an automated simulation-based verification of formal specification refinement proposed in [10]. The disadvantage of this verification approach is that it is essentially not complete, due to informal verification, and therefore some errors may not be revealed. Our approach is based on formal verification; thus, it provides higher assurance of the result.

III. Abstraction Refinement in HSSL

HSSL [1] is intended for ESL behavioral specification and modeling of reactive systems. It supports behavioral specification concepts, such as hierarchical decomposition, refinement, reuse, timing, exception handling, concurrency, and synchronization.

The top-level structure in HSSL is a *system*, which has *input* and *output* variables (representing an interface with the system environment), the *state variables* (representing an internal state), and the sets of *agents* and *processes* specifying the system behavior. An *agent* describes the communication of the system with the environment between two states. Such a communication is specified by a *communication formula* defined by a set of *actions*. A single *action* specifies the state of the system inputs and outputs in a specific time stated by an event. A *process* represents an execution order of multiple agents, which enables sequential or concurrent composition of agents, loops, interrupts, and conditions.

The refinement process enabled in HSSL is defined in [11]. In this process, the refined specification is understood as an implementation of the previous specification. It can contain multiple stages (i.e. partial refinements), which always include a creative transformation leading to the refined specification and the verification step checking whether the refined specification corresponds to the previous more-abstract specification. The refinement continues until the specification contains a sufficient amount of details and the high-level synthesis can proceed (see Fig. 1).

There are two kinds of refinement defined in HSSL:

1. *Agent to process refinement* – The complex agent is replaced by a process that composites multiple less complex agents. Each of these agents specifies a simpler partial behavior. The introduction of a new process implies new state transitions between new control states, which can be represented by some additional state variables.

2. *Communication refinement* – The input/output variables and the communication sets are replaced by more detailed constructs. This kind of refinement is mostly used when a new communication protocol is introduced to communicate with the system environment. Usually, new input/output variables are added to the specification.

The functional verification using formal equivalence checking depends on the used kind of refinement. In case of the first kind (i.e. agent to process), the introduced process is transformed to a single agent that combines the partial behaviors specified by the agents in the process. The equivalence from the side of environment is checked by comparing of the communication sets and the final states of the two agents (the original agent before the refinement step and the agent representing the process after the refinement). In case of the second kind of refinement (i.e. communication), a proper mapping between the old and the new input/output variables in the agent must be found, which impacts the communication set as well as the final state of the agent. A so-called reduction function [11] is applied to the refined agent and then the equivalence to its previous version is verified.

IV. HSSL Verification Tool

HSSL Verification Tool is a tool for verification of whether the refined system specification in HSSL is equivalent to the original one. Thus, it is verified whether the refinement procedure was correctly processed.

The requirements for the tool have been stated as follows.

- Loading and understanding of HSSL specification.
- Transformation of specification into a suitable internal format.
- Checking the equivalence between two loaded and transformed specifications.
- Clear visualization of HSSL source code.
- User-friendly graphical user interface.

The first requirement is fulfilled by implementation of a HSSL parser that can analyze HSSL source code, in which a system is described. A suitable object-based internal format is developed for easier manipulation with the specification. The equivalence checking is the key feature offered by the tool. It compares the two transformed specification models, while it takes into account specific rules of the refinement process. The tool can also serve as a development environment, in which the source code can be written. For a clear visualization, two separated side-by-side windows for the two specifications are suitable. And finally, a list of language constructs and their hierarchy based on the specified digital system is provided for the designer's convenience.

A. *Verification Tool Design*

When designing a software product it is important to determine what functionality it should contain. Since a HSSL specification is the input to the tool, there needs to be one part of the tool focused on HSSL analysis. It is necessary to develop a method for translation of the written text constructs of HSSL to data objects in programming language in order the developed tool to handle them. The most important part of the tool functionality is the ability to formally verify the HSSL specification. From the implementation perspective, the tool is divided into three parts – analyzing part, storage part, and comparison part. Message exchanges between these parts are driven by user interface. These parts represent the key independent components of the tool, as illustrated in Fig. 2.

User Interface enables interaction between the designer and the underlying verification functionality of the tool. It provides capability to load two files with HSSL specifications and clearly displays the verification result.

HSSL Analyzer provides analysis of the HSSL specification. It is a difficult task; therefore, it is divided into three internal specialized modules – lexer, parser, and compiler.

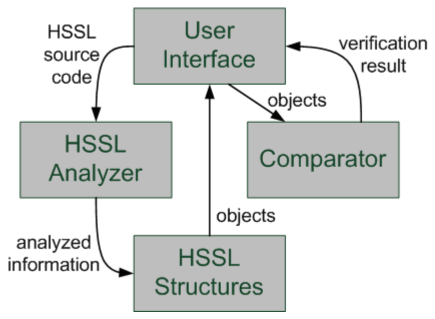


Figure 2. The proposed verification tool architecture

HSSL Structures contains multiple classes that represent the HSSL constructs. The source-code text is transformed into objects of these classes for easier manipulation during the comparison. These classes include the following.

- *HSSL System* – this is a representation of the overall system model specified in HSSL.
- *Agent* – is a fundamental building block of HSSL specification, representing finite partial behavior.
- *Action* – describes the communication of the agent with the environment during its execution.
- *Process* – describes the composition of multiple agents to execute sequentially or in parallel.
- *Communication instance* – contains all the possible components of a communication set.

Comparator is used to actually verify the specification. It takes into account the rules of the abstraction-refinement process developed for HSSL specification [11].

The HSSL analysis has to recognize not only the text (syntax) but also the semantics. A commonly used procedure for such analysis capabilities is parsing. It is a lexical-syntactic analysis of the text, during which the text is analyzed and the sequences of keywords create symbols (tokens). To meet the requirements, the proposed tool uses techniques very similar to parsing. The text analysis is conducted in the following way.

1. Finding the specific keywords.
2. Based on the found keywords, objects are created in the programming language.
3. Individual objects are linked by logical connections in the text.

After the analysis of the specification is complete, the tool will have a hierarchy of objects usable for the processing. We have used the object-oriented implementation language C#, which makes object manipulation easier than manipulation of the text alone.

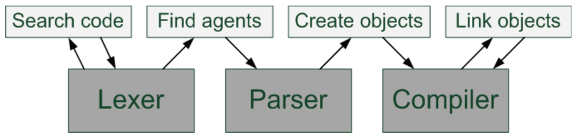


Figure 3. The proposed analyzer modules and actions

The proposed actions and communications between the modules of *HSSL Analyzer* (lexer, parser, and compiler) are illustrated in Fig. 3.

After the necessary objects are created based on the lexical analysis, these objects are passed to the verification part of the tool (*Comparator*) and the automated verification process begins. The proposed verification process consists of four basic steps, which verify the interfaces, the organizational structures, the communication, and the functionality.

1) Comparison of inputs and outputs

In this step the tool verifies whether the input and output variables (representing system interfaces) are equivalent in both specifications and whether the types of variables (i.e. value domain) do not contradict each other.

2) Pairing of agents and processes

This verification step checks whether each process or agent in one specification has the corresponding form in the other specification. An unpaired process or agent means that the specifications are not equivalent – i.e. one of the specifications contains the functionality not specified in the other one.

3) Comparison of communication sets

The communication sets of the agents are concatenated into the final communication set based on the predefined rules [11]. This set is then compared to the paired agent/process in the refined specification. If some communication word is not equivalent in the compared specifications, the entire specification is declared to be non-equivalent.

4) Comparison of final states

The final states of the processes are deduced based on the sequences of the concatenated agents. The final state represents a logical result of the system partial functionality. This verification step actually involves checking whether the final states of the paired processes are equivalent.

The proposed automated verification method does not verify actual values (like in case of the simulation), but instead, it is based on static analysis and uses formal logic of process algebra to prove or disprove equivalence. Thus, the verification is very fast and the eventual result is complete.

B. Tool Restrictions

We have developed HSSL Verification Tool as a prototype to prove that the used verification process can be automated. Therefore, there are some restrictions that can be eliminated in a further version of the tool. These restrictions represent some coding style rules, which enable faster code analysis and pairing of HSSL constructs.

- Writing of “if-else” constructs must be used in a form with parentheses. Example:
  if (condition) (Agent1) else (Agent2.Agent3)

- This restriction also applies to the other branching and cyclic constructs, such as “while”, “do-while”, “switch” and “loop”.

- Processes that describe the same behavior must have the same identifier. If an agent in the original specification is refined, the extended process must retain his identifier. Example:
  Agent = ProcAgent

- Input and output variables must have the same identifier and type (i.e. domain) in both specifications. The order of variables specification is significant.

Keeping in mind these restrictions, the designer can utilize the verification capabilities which this tool offers. Because of these coding-style restrictions, the tool is especially suitable for verification of the refined specification against its original form.

V. Experimental Testing

The application testing has been performed upon the exemplar specifications, into which artificial errors were inserted. If the developed tool detects an error, it stops the verification process and shows the error. Each error is identified by the error code and the error object (if possible). HSSL Verification Tool can detect the errors described in Table 1.

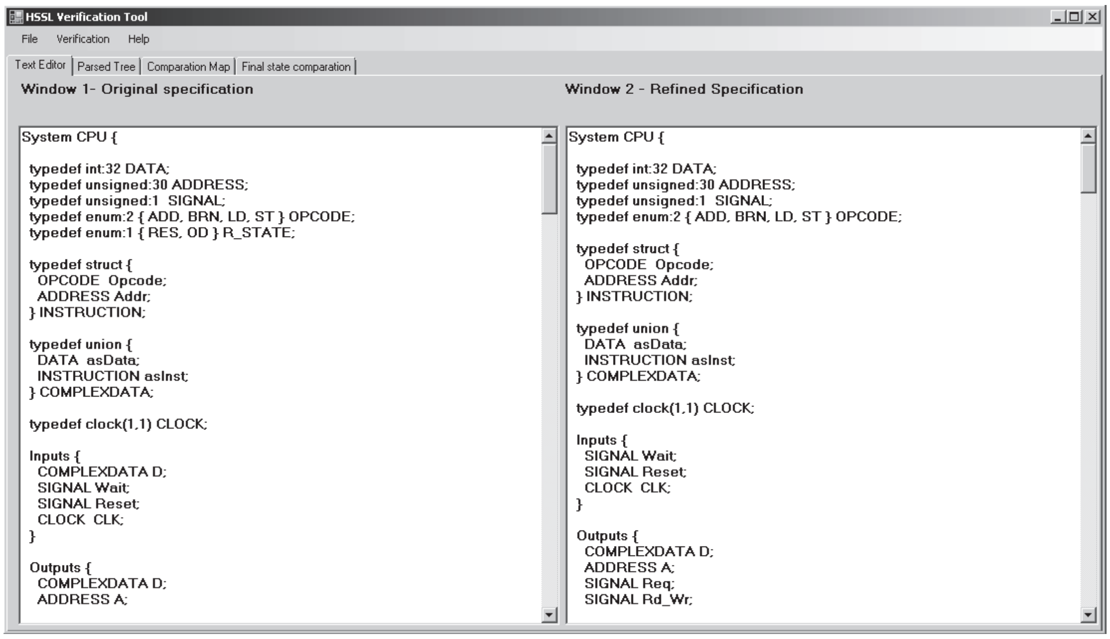
We provide an example of one specific error detection testing in the following text. It is aimed towards a missing variable kind of an error. The used HSSL specifications describe a simple processor in the coarse (original) form and the refined form.

The user interface, provided in Fig. 4, illustrates the original specification on the left which contains an input variable that is not present in the refined specification on the right.

The developed tool has loaded and analyzed both specifications correctly. After starting the verification, the tool has displayed the error code 71 with the message “In original specification is unpaired input”. It has also identified that the erroneous object is the input variable

TABLE I.
Description of Errors According to Error Codes

Error Code	Error Description
21	A different number of communication sets.
22	An unpaired communication word.
23	An unpaired communication word in the original specification.
24	An unpaired communication word in the refined specification.
31	The final states are not equivalent.
32	An unpaired state in the original specification.
33	A missing state in the refined specification.
51	An unidentifiable object found during compilation.
71	An unpaired input in the original specification.
72	An unpaired variable in the original specification.
73	An unpaired output in the original specification.
81	An unpaired agent in the original specification.
82	An unpaired agent in the refined specification.
90	An unknown error.



Window 1- Original specification

```
System CPU {

  typedef int:32 DATA;
  typedef unsigned:30 ADDRESS;
  typedef unsigned:1  SIGNAL;
  typedef enum:2 { ADD, BRN, LD, ST } OPCODE;
  typedef enum:1 { RES, OD } R_STATE;

  typedef struct {
    OPCODE  Opcode;
    ADDRESS Addr;
  } INSTRUCTION;

  typedef union {
    DATA  asData;
    INSTRUCTION asInst;
  } COMPLEXDATA;

  typedef clock(1,1) CLOCK;

  Inputs {
    COMPLEXDATA D;
    SIGNAL Wait;
    SIGNAL Reset;
    CLOCK  CLK;
  }

  Outputs {
    COMPLEXDATA D;
    ADDRESS A;
```

Window 2 - Refined Specification

```
System CPU {

  typedef int:32 DATA;
  typedef unsigned:30 ADDRESS;
  typedef unsigned:1  SIGNAL;
  typedef enum:2 { ADD, BRN, LD, ST } OPCODE;

  typedef struct {
    OPCODE  Opcode;
    ADDRESS Addr;
  } INSTRUCTION;

  typedef union {
    DATA  asData;
    INSTRUCTION asInst;
  } COMPLEXDATA;

  typedef clock(1,1) CLOCK;

  Inputs {
    SIGNAL Wait;
    SIGNAL Reset;
    CLOCK  CLK;
  }

  Outputs {
    COMPLEXDATA D;
    ADDRESS A;
    SIGNAL Req;
    SIGNAL Rd_Wr;
```

Figure 4. Graphical user interface

"D". An illustration of the verification result notification is provided in Fig. 5.

The tool has correctly identified all the tested errors and the verification result was correct in all test cases.

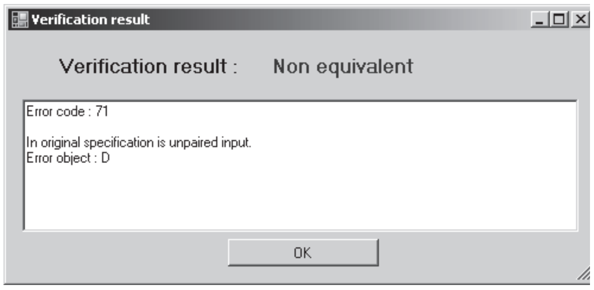
VI. Conclusion

In this paper, we have proposed a verification tool for refined specification in HSSL. This tool is unique in providing automated formal-verification capabilities to the system development process involving HSSL modelling.

We have described the requirements for the tool and we have proposed a solution fulfilling these requirements. The developed verification tool compares two specifications in HSSL, where one is the refined form of the other. The proposed verification process is based on the lexical analysis of the specifications, the creation of objects and their logical linking into a tree, and checking the equivalence between the two object-tree representations corresponding to the two specification stages. Using experimental evaluation, we have carefully tested the detection capabilities for specific errors. The detected error in the specification along with the erroneous object is clearly identified.

The proposed HSSL Verification Tool enables the designer to quickly correct the issue and to keep specification equivalent during the abstraction-refinement process. It is especially suitable for novice designers (such as students), for which the manual formal verification process is too complicated and error-prone, or just takes too much time. Therefore, it is valuable support in the education process. It can be also used by a teacher to quickly evaluate high amount of students' assignments. This tool supplements the set of existing design-automation tools supporting the HSSL modelling. The modular design of the tool enables easy extensibility and makes it suitable to be used as a part of a more complex development environment.

Functionality of the tool could be extended in several ways, such as checking of the HSSL syntax or involvement of new HSSL features during verification (e.g. power-management specification [4]). These extensions possibilities represent potential focus of our further work in this area.



Verification result : Non equivalent

```
Error code : 71

In original specification is unpaired input.
Error object : D
```

Figure 5. The verification result notification

Acknowledgment

This paper was created with the support of the Ministry of Education, Science, Research and Sport of the Slovak Republic within the Research and Development Operational Programme for the project "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the European Regional Development Fund. This work was also partially supported by the Slovak Scientific Grant Agency (VEGA 1/0616/14 and 1/0836/16) and the Slovak Research and Development Agency (APVV-15-0789).

References

[1] N. Fristacky, J. Kacerik, T. Bartos, and M. Kardos, "Behavioral specification model and language for digital systems," Tech. Rep., Slovak University of Technology, 2000.

[2] N. Fristacky, J. Kacerik, and T. Bartos, "A mixed event-value based specification model for reactive systems," in *SoC Methodologies and design Languages*, Springer US, 2001, pp. 193-204.

[3] P. Majtáz, "Formálna špecifikácia vstupného system," Master thesis, University of Zilina, 2001.

[4] D. Macko and K. Jelemenská, "Managing digital-system power at the system level," in *IEEE Africon 2013 Sustainable Engineering for a Better Future*, 2013, pp. 179-183.

[5] S. Sidor, “Simulation of digital system specification in HSSL”. Master thesis, Slovak University of Technology in Bratislava, 2013.

[6] K. Jelemenská, M. Kardoš, and P. Čičák, “HSSL specification high-level synthesis,” in *2015 13th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2015, pp. 171-176.

[7] Y. Kawamata, C. Sommer, F. Ishikawa, and S. Honiden, “Specifying and Checking Refinement Relationships in VDM++,” in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, 2009, pp. 220-227.

[8] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, “Automated Verification of AADL-Specifications Using UPPAAL,” in *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE)*, 2012, pp. 130-138.

[9] T. Peng and G. Ding, “Formal Specification and Automated Verification of UML2.0 Sequence Diagrams,” in *2012 IEEE International Conference on Granular Computing (GrC)*, 2012, pp. 370-375.

[10] S. Yamada, A. Keijiro, S. Kusakabe, and Y. Omori, “Validation of stepwise refinement with test cases generated from formal specification,” in *TENCON 2010 - 2010 IEEE Region 10 Conference*, 2010, pp. 2449-2453.

[11] J. Kačerík, “Behavioral specification refinement in HSSL language environment,” in *Proceedings Sig-VHDL & ECSI of “Forum on Design Languages” FDL'2000*, 2000, pp. 329-336.