Validation of Contract Decomposition by Testing

Mass Soldal Lund

Department of Informatics, University of Oslo, Norway
SINTEF Telecom and Informatics, Norway
email: mass.s.lund@sintef.no

Abstract

In development of concurrent system there is a need for specification techniques that handle modularity and integration. Contract oriented specifications from the field of formal methods offer modularity, but are hard to use. We present a scheme for making contract oriented specifications in the graphical specification language SDL. Further we propose a testing strategy for validating decomposition of contract oriented specifications based on the composition principle, which may be used for testing integration at specification level. The strategy was implemented in a prototype tool for testing of decomposition of SDL contract specifications, and an empirical study showed that this prototype was considerably more efficient than a prototype based on a conventional strategy.

1 Introduction

Reasoning about composition of parallel components is a non-trivial business, and has been subject to considerable research in the field on formal methods.

In [2] Martín Abadi and Leslie Lamport present a rule for validating decomposition of contract oriented specifications[1] by logical proofs, called the Composition Theorem. The rule may in certain respects be seen as a generalization of similar rules in [15] and [18]. We believe the Composition Theorem can be generalized to a universal principle for validation of contract decomposition, which will be referred to as the *composition principle*. Results from [5] and [19] support this belief.

Formal methods are however little used in real-life system development [8, 10]. Some of the reason may be that formal techniques are tedious and time-consuming, and people without thorough training find them hard to use [9].

In [17] we propose a testing strategy for validating decomposition of contract oriented specifications based on the composition principle. The strategy can be implemented in a tool that does automated testing of whether a decomposition of a contract specification for a system or component, into a network of contract specifications for sub-components, is valid. For several reasons we believe such a tool may have a mission in system development.

In the contract oriented paradigm, both the components and their requirements on their environments are specified. This provides a modularity to specifications which may ease development and maintenance of concurrent system. During development contract oriented specifications may prevent unexpected behavior of components because the context in which the components are expected to work is explicitly specified. System maintenance

[1] Contract oriented specifications are also called assumption/guarantee, assumption/commitment and rely/-guarantee specifications. The expressions contract specifications and contracts will also be used in this paper

may benefit from contract oriented specifications because they make it easier to figure out the impact on the rest of the system if one or more components are changed or replaced.

Integration is a crucial part of development of concurrent systems. The proposed testing strategy can be used in iterations to ensure consistency between specifications of different detail, and thereby resolve complications in integration at specification level, i.e., before implementation.

Reasoning about parallel components is complicated because a component and its environment (consisting of other components) may be mutually dependent upon each other, and this may lead to circularities in the reasoning.

The strategy has been implemented in an experimental prototype for testing of decomposition of SDL[2] specifications written in a contract oriented style. This tool is referred to as the *Indirect testing tool*. As a reference a prototype tool based on a conventional testing strategy, direct testing of decompositions, was also implemented; the *Direct testing tool*.

While the Indirect testing tool does testing according to the premises of the composition principle, the Direct testing tool does direct testing of the conclusion. This is illustrated below.

$$\left.\begin{array}{l} p_1 \\ p_2 \\ \vdots \\ p_n \end{array}\right\} \text{ Indirect testing tool}$$

$$\overline{\qquad}$$

$$c \qquad \text{Direct testing tool}$$

In an empirical study of the two tools, it was shown that the Indirect testing tool is considerably more efficient than the Direct testing tool, without doing tests of lower quality.

Section 2 provides some background on the semantics and notation used in this paper. Section 3 explains how the composition principle is related to SDL and provides details on the tools. Section 4 provides the results from empirical testing of the two tools, and finally, in section 5, some conclusions are presented.

2 Background

The testing strategies are founded on the stream-based semantics of the FOCUS method [4]. In this formalism, components communicate by the means of sending streams of messages over directed channels. Discrete time is represented by time ticks which divide the streams into time units. An infinite timed stream is a sequence of an infinite number of ticks and possibly an infinite number of messages. Each tick represents the end of a time unit, and for all $t \in \mathbb{N}$ there can be arbitrary finite number of messages between the t'th and the $(t+1)$'th tick. A finite timed stream always ends with a tick, i.e., a timed stream can never be truncated in the middle of a time unit. If s is a stream, $s{\downarrow_t}$ denotes s truncated after t ticks.

If D is a set of messages, D^∞ denotes the set of all infinite timed streams with messages from D, D^* denote the set of all finite timed streams with messages from D, and $D^\omega = D^\infty \cup D^*$ the set of all streams with messages from D.

Transmission of a message m over a channel c is instantaneous, which means that the output of m and the input of m happens within the same unit of time.

The behavior of a system or a component is defined as a relation between the input streams and output streams of that system or component. Since this is a binary relation, it

[2] Specification and Description Language [14]

defines a set of pairs of tuples of streams. This set may be called a *input/output relation* and its element are referred to as *input/output pairs*.

Based on the work done in [11] and [12] a subset of SDL[3] is formalized in the FOCUS semantics. In this formalization, an SDL specification S is represented by a binary predicate $[\![S]\!]$ over tuples of streams where the first tuple is the input stream and the last tuple the output streams:

$$[\![S]\!] : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B} \tag{1}$$

$[\![S]\!]$, called the *denotation* of S, is the input/output relation of S mentioned above. Sets and predicates are essentially the same. The notions of sets and predicates are in the following used interchangeably, and both set operators and logical operators are applied on denotations after convenience.

A specification S' is said to refine a specification S iff $[\![S']\!] \subseteq [\![S]\!]$ (alternatively $[\![S']\!] \Rightarrow [\![S]\!]$), and this is denoted by a special refinement relation: $S \rightsquigarrow S'$. This definition correspond to behavioral refinement in [4], but does for example not apply to refinement of interface or communication. The use of the subset relation make sure that S' has all the behaviors of S, but allow S' to be more detailed (and closer to implementation) than S.

In textual notation composition is denoted by the composition operator $\otimes$. The composition operator is semantically defined as

$$[\![S_1 \otimes S_2]\!] \stackrel{\text{def}}{=} \exists l \in (D^\omega)^k : [\![S_1]\!] \wedge [\![S_2]\!] \tag{2}$$

where l is a tuple of streams over the internal channels of the composition. Since it can be ensured that the composition operator is commutative and associative, there is no need for defining composition of more than two components.

Details on the SDL subset and the formalization can be found in [17].

2.1 Contract Oriented Specifications

In the contract oriented style a component C is specified by an assumption A and a guarantee G. Together they form the specification S of C which is denoted $S = (A, G)$.

Contract oriented specifications are based on the principle that a specified component should fulfill the guarantee as long as the assumption is met. Informally the relationship between A and G is:

- A describes the behavior of the relevant part of the environment under which C is required work as specified.
- G describes the behavior guaranteed by C when the environment behaves according to A.

The environment of a component consists of other components in the system it belongs, and possibly the external environment of the whole system.

A specifies the relevant part of C's environment, which means that A specifies the environment from C's point of view. In practice A will in this approach mainly specify the format of the input to C.

When reasoning about parallel components, there is a risk of getting circularities because the components may be mutually dependent upon each other. In order to avoid circularities in the reasoning, we follow [2] and add the requirement to the contract specifications

[3]The subset is contained in both SDL-92, SDL-96 and SDL-2000

that the guarantee G of a specification S must hold as long as the assumption A holds and at least one step longer. In the semantics based on timed streams, this step is equal to one time unit.

When $S = (A, G)$ is formalized, A and G are defined as predicates over timed streams of type D^ω:

$$A : (D^\omega)^m \times (D^\omega)^n \to \mathbb{B} \tag{3}$$

$$G : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B} \tag{4}$$

where the component has n input channels and m output channels. Since A specify the assumed behavior of the environment of G, the output of G is the input of A and vice versa. For a contract specification $S = (A, G)$ a special denotation $[\![S]\!]$ is defined as

$$[\![S]\!] \stackrel{\text{def}}{=} \forall t \in \mathbb{N} : A(o\downarrow_t, i\downarrow_t) \Rightarrow G(i\downarrow_t, o\downarrow_{t+1}) \tag{5}$$

where i is the tuple of input streams and o the tuple of output streams. This definition captures both the requirement that the guarantee should hold as long as the assumption holds and the "one step longer" requirement.

When we specify contracts in SDL, the assumption A and the guarantee G are both specified by SDL processes. As a starting point, A is specified in a simple form where it only shows which messages the environment is allowed to send over which channels when in a given state.

3 Testing Tools

The purpose of the tools is to validate refinements

$$S \rightsquigarrow \otimes_{j=1}^{n} S_j \tag{6}$$

where S and S_j, for $j = 1, 2, \ldots, n$, are contract specifications.

Both testing tools do validation of refinements by means of well-known methods for functional testing [3]. In difference from normal testing, a specification is not tested against an implementation (or an implementation tested against a specification), but against another specification of the same system at higher level of abstraction.

The prototype tools use functionality for generating and executing test cases from the specifications tool Telelogic Tau SDL Suite [20]. Aside from the SDL editor the part of Telelogic Tau used is SDL Validator. SDL Validators are executable programs generated from an SDL specification with the ability to simulate the specific SDL specification from which they are generated.

As test case format, Message Sequence Charts (MSCs) [13] is used. SDL Validator has functionality for generating MSC test cases from an SDL specification and for verifying that an MSC is a possible execution of an SDL specification. Both generation and executing MSC test cases are done by specialized state space exploration algorithms [7, 16].

The MSCs generated by SDL Validator are simple; they only contain instances and messages. The SDL specification from which they are generated is represented by one single instance, and the external channels of the specification are represented by one instance each. The SDL subset is restricted to only have directed channels, so the MSC test cases can easily be related to the input/output pairs of the semantics.

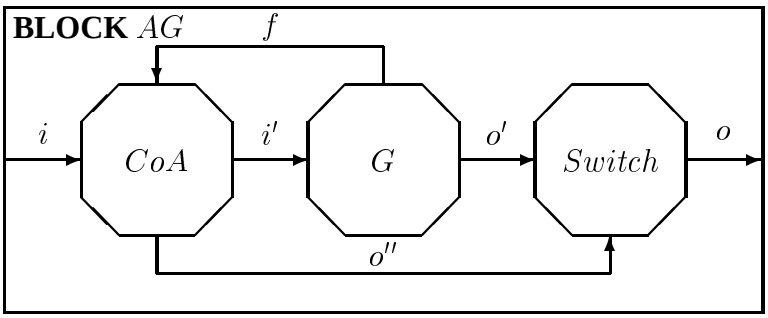

Figure 1: SDL block for contract specifications

3.1 Direct Testing Tool

The direct testing tool is based on testing the subset relation

$$[\![\otimes_{j=1}^{n} S_j]\!] \subseteq [\![S]\!] \tag{7}$$

directly. There is then a need for representing $S = (A, G)$ in SDL. In the functional setting the interpretation is made that if the assumption of a component is violated at time t, the component starts producing chaos at time $t + 1$. However note that one possible chaotic behavior is to produce correct output, so the component starts behaving chaotically *at least* one time unit after the assumption is violated.

$S = (A, G)$ is specified by a special SDL block AG, shown in figure 1[4]. Instead of A simulating the environment, an SDL process CoA that recognizes legal input is used. The reason for this is a pragmatic one; the testing tool we built to do functional testing of SDL contracts needed the channel i from the boundary of the SDL block to CoA in order to monitor the input. This channel was also necessary when composing AG blocks.

In addition to recognizing legal (assumed) input, CoA is the part of the specification that does the chaos production in case the assumption is violated. As long as CoA receives legal input, the input is forwarded to G over i', but if CoA receives unexpected input it ignores all later input and starts outputting chaos on o''. CoA is a straightforward modification of A where the implicit transition is overridden by a transition to a state $chaos$ from which random output is spontaneously output.

The $Switch$ component forwards the output of G as long as the assumption is not violated, but switches to forwarding the output of CoA if CoA starts producing chaos. In order to model the "plus one"-semantics, $Switch$ has a delay of one time unit before conducting the switch, i.e., if CoA is violated after t time ticks, $Switch$ forwards messages from G until time tick $t + 1$ before switching.

CoA and $Switch$ are interpreted to be instantaneous. Intuitively this is a reasonable assumption, since they are simple components that would not use any considerable amount of computational time. This assumption does however not hold in our implementation, but as we show below this was not a problem. The guarantee G is assumed to have a delay of at least one time unit. It is reasonable to assume that there is some delay due to use of computational time in the component.

In AG there is also a channel f from G to CoA. This is the *feedback channel*, and is used if we want to specify that the environment responds to output from the component. Since this possibility is omitted, the channel f is neither used nor discussed and appear only

[4]Figure 1 show the AG block of a component with one input channel and one output channel, but this scheme can easily be generalized to specification of components with more input and output channels

as an illustration of how the extension to SDL contract specifications with feedback could be done.

In [17] we prove the following propositions:

Proposition 1 *If (A, G) is a contract specification where A and G are safety properties, AG is an SDL representation of (A, G), and the assumption that CoA and $Switch$ are instantaneous is true, then*

$$[\![AG]\!] = [\![(A, G)]\!]$$

Proposition 2 *If AG and (A, G) are as in Proposition 1, except that CoA and $Switch$ have additional delay, then*

$$[\![AG]\!] \subseteq [\![(A, G)]\!]$$

3.1.1 Testing

When the refinement (3) is tested, SDL Validators are generated from the SDL specifications S and $\otimes_{j=1}^{n} S_j$. Test cases are automatically generated from $\otimes_{j=1}^{n} S_j$ and verified (executed) against S.

If all test cases generated from $\otimes_{j=1}^{n} S_j$ are verified by S, we have some evidence that the refinement, i.e., the decomposition, is valid. If one or more of the test cases generated from $\otimes_{j=1}^{n} S_j$ is not verified by S, the decomposition is incorrect.

3.2 Indirect Testing Tool

The Indirect testing tool is based on a simplification of an instance of the composition principle composition rule formulated in [19]. This simplified composition rule is formulated in Proposition 3. The soundness of Proposition 3 is proved in [17].

Proposition 3 *If $S = (A, G)$ and $S_j = (A_j, G_j)$, for $j = 1, 2, \ldots, n$, then*

$$\frac{\begin{array}{l}\forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge \bigwedge_{j=1}^{n} G_j(i_j{\downarrow_t}, o_j{\downarrow_{t+1}}) \Rightarrow A_k(o_k{\downarrow_{t+1}}, i_k{\downarrow_{t+1}}), \\ \qquad \textit{for } k = 1, 2, \ldots, n \\ \forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge \bigwedge_{j=1}^{n} G_j(i_j{\downarrow_t}, o_j{\downarrow_{t+1}}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})\end{array}}{S \rightsquigarrow \otimes_{j=1}^{n} S_j}$$

In order to do testing, the premises of Proposition 3 are translated to SDL specifications. With the interpretation of composition as logical conjunction (in definition (2)), the SDL specifications $LS = A \otimes (\otimes_{j=1}^{n} G_j)$, $RS_{1.k} = A_k$ and $RS_2 = G$ are made. LS, $RS_{1.1}$ and RS_2 for an example where S is decomposed into two components with common (internal) channels l_1 and l_2 are shown in figures 2-4.

In LS and $RS_{1.k}$, the assumptions A and A_k are modified into A^{LS} and $A^{RS}_{1.k}$. A^{LS} recognizes and forwards legal input in the same manner as CoA, but instead of producing chaos if illegal input is received sends a special message err over channel o' which indicate that the assumption is violated. Before the test cases are executed the illegal input, err and all input messages after err are removed. In this way the constraints on the streams in Proposition 3 are ensured. All test cases then contain legal input until a point in time t and all legal output produced from this input, i.e., legal output until at least $t + 1$.

Since the $A^{RS}_{1.k}$ are used for verifying and not generating test cases, each $A^{RS}_{1.k}$ is a modification of A_k which simulate the component's environment by spontaneously outputting legal input.

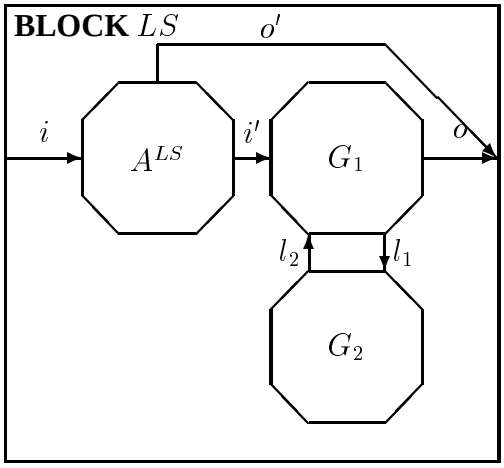

Figure 2: **BLOCK** LS

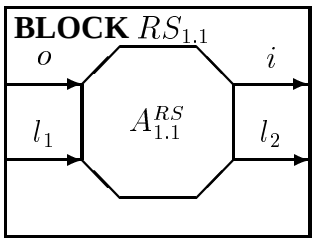

Figure 3: **BLOCK** $RS_{1.1}$

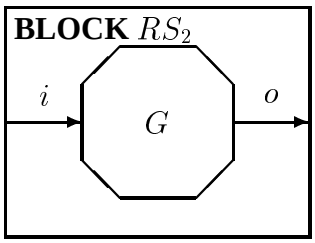

Figure 4: **BLOCK** RS_2

3.2.1 Testing

SDL Validators are generated from the SDL specifications LS, $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and RS_2. Test cases are generated from LS and executed against all $RS_{1.k}$ and RS_2.

If all test cases generated from LS are verified by each $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and RS_2, there are some evidence that the decomposition is valid. If one of the test cases generated in testing of one of the premises is not verified, the decomposition is invalid.

4 Empirical Results

In order to validate and compare the two testing strategies, 15 example decompositions were tested by both tools.

The examples are artificial in the sense that they were constructed in order to conduct these tests; they are not examples from real system development and are fairly simple compared to what we may find in real-life system development, both in respect to complexity and size (an average of 9.2 control states per example).

Even though the examples are artificial we believe they cover different situations that may occur in contract decompositions, and most of them are dedicated to test specific features of the testing strategies.

We do believe they are representative for a considerable class of contract decompositions, and that we should be able to generalize from the results.

4.1 Correctness

According to the theory, there are two ways of making correct behavioral refinements of a contract oriented specification [1, 4, 6]:

- Strengthening the guarantee, i.e, making it more specific (less general).
- Weakening the assumption, i.e., making it more general (less specific).

If we analyze the examples in accordance to this with a black box view of the composite specifications resulting from the decomposition, each of the examples can be placed in one of six groups:

Group	**1**					**2**			**3**
Example	1	2	3	12	14	5	6	13	8
Valid	Y	Y	Y	Y	Y	Y	Y	Y	Y
Direct validation	Y	Y	Y	Y	Y	Y	Y	Y	Y
Indirect validation	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 1: Result of testing groups 1-3

Group	**4**			**5**		**6**
Example	7	10	11	4	15	9
Valid	N	N	N	N	N	N
Direct validation	N	N	N	N	N	N
Indirect validation	N	N	N	N	N	N

Table 2: Result of testing groups 4-6

1. Neither the assumption nor the guarantee were neither strengthened nor weakened; examples 1, 2, 3, 12 and 14.

2. The assumption was neither strengthened nor weakened, while the guarantee was strengthened; examples 5, 6 and 13.

3. The assumption was weakened, while the guarantee was neither strengthened nor weakened; example 8.

4. The assumption was neither strengthened nor weakened, while the guarantee was weakened; examples 7, 10 and 11.

5. The guarantee was neither strengthened nor weakened, while the assumption was strengthened; examples 4 and 15.

6. The assumption was weakened, while the guarantee was changed in a way that may be seen as strengthening and weakening at the same time; example 9.

The results of validating the examples with the testing tools are presented in tables 1 and 2.

We see that the testing tools gave the same results with respect to validation for all 15 examples, i.e., the testing tools validated exactly the same examples. According to the theory the examples of groups 1-3 are valid and examples of groups 4-6 are invalid, which means both tools validated exactly the valid examples.

4.2 Efficiency

When analyzing the efficiency of the tools, another grouping of the examples are used:

1. Examples where the assumption of the decomposition is not equivalent to **true**[5]; examples 1, 2, 3, 4, 10, 14 and 15.

2. Examples where the assumption of the decomposition is equivalent to **true**, but the assumption of the original specification is not; examples 8 and 9.

[5]By equivalent to **true** is meant that the assumption allows all type correct input from the environment

	Example	1	2	3	4	10	14	15
	Valid	Y	Y	Y	N	N	Y	N
	No. states	7	15	20	10	14	9	7
Direct	Exec. time	7:25	6:33	11:37	10:08	7:06	0:48	6:57
	Test cases	10	15	14	10	10	5	10
	Av. length	94.4	1.5	121.2	74.8	117.9	41.4	116.4
	Coverage	95.2	70.0	96.1	97.6	88.7	96.4	100.0
Indirect	Exec. time	0:11	0:16	0:20	0:14	0:13	0:12	0:12
	Test cases	2	6	5	4	2	2	2
	Av. length	4.0	4.3	5.6	2.0	8.0	3.0	4.0
	Coverage	100.0	100.0	97.9	100.0	100.0	100.0	100.0

Table 3: Efficiency of testing tools

3. Examples where both the assumption of the decomposition and the assumption of the original specification are equivalent to **true**; examples 5, 6, 7, 11, 12 and 13.

The interesting examples are all in group 1, because assumptions equivalent to **true** make contracts superfluous and because in this group differences in efficiency between the tools were observed.

In order to find any differences in the efficiency of the two testing tools, the time used for generating and executing test cases and the symbol coverage when test cases were executed (the fraction of the SDL symbols covered after executing the set of generated test cases) were measured. In addition states in the examples were counted as a measure of size, generated test cases were counted and messages in test cases were counted as a measure of length. The results from group 1 are found in table 3. From the table we observe:

- The Direct testing tool had much longer execution times than the Indirect testing tool.
- The Direct testing tool generated more and longer test cases than the Indirect testing tool.
- The Direct testing tool gave some lower symbol coverage than the Indirect testing tool.

We see a clear trend that the Indirect testing tool is more efficient than the Direct testing tool, with both respect to the use of time and the number and length of test cases. The reason for the differences is that the Direct testing tool used considerably time on generating test cases with chaos.

We also see that there are no loss in symbol coverage from the Direct tool to the Indirect tool. When we in addition know that the tools validate exactly the same examples, we can conclude that the quality of the tests conducted by the Indirect testing tool is not lower than the quality of the tests conducted by the Direct testing tool. It is hard to see any effect of the size of the examples, neither can we see any effect caused by validity.

5 Conclusions

We have, via formalization in the stream-based FOCUS semantics, showed how contract oriented specifications may be represented in SDL. Further we have showed how the

stream-based semantics may be related to functional testing and how the composition principle can be used as a testing strategy for validating contract decomposition. Details on the formalization and proofs can be found in [17].

The proposed testing strategy can be implemented as a testing tool where the input is specifications of components consisting of assumptions in some standardized form, in addition to guarantees and the connections between components. In the prototype tool the modifications to assumptions were done manually. These modifications, and the setup of the specifications under test, are quite straightforward and schematic and can be automated without much trouble.

In construction of the prototype, several restrictions were made. This means the set of specifications the tool was able to handle is a proper subset of the specifications expressable by the semantics. There are however nothing in the analytic results that indicate that a full implementation is not possible.

In the prototype, SDL was used as specification language. However the only real requirement on the specification language is that it is executable. Another natural choice of specification language could be the Unified Modeling Language (UML). Assumptions and guarantees would then be specified with UML state machines, and the connection between components could typically be specified with a combination of class diagrams and object diagrams.

Other possible specification languages could be traditional programming languages, e.g. Java, since programming languages most certainly are executable. One could even imagine a tool that supports different specification languages, for example a tool that supports both UML and Java. This tool could be used throughout iterative system development processes. In early iterations, UML specifications could be decomposed into other UML specifications, and in the later iterations Java implementations of components could be integrated in the testing. A tool that supports both UML and SDL could be used in system development processes where UML is used for high level specifications and SDL for lower level specifications and system design.

There are no reasons why the assumption part and the guarantee part of a contract specification have to be specified in the same specification language. It is for example possible to imagine that standard components implemented with Java could be delivered with assumptions written in UML.

The empirical results show that the two prototypes gave the same results with respect to validation, and that the results were in accordance to the theory of behavioral refinement. They also show that the Indirect testing tool is more efficient than the Direct testing tool when the decomposition of a specification has assumptions that are not equivalent to **true**, which in this setting is the interesting situation, and that the Indirect testing tool does not do tests with lower quality than the Direct testing tool.

Both the analytic and the empirical results support the thesis that a tool based on the proposed testing strategy may have a future. The thesis that contract oriented specifications have a mission in system development has however only been used as motivation and is not explicitly addressed. A possibility for further work could be to realize a prototype of the tool that uses UML as specification language and conduct a case study within a real-life system devlopment where the tool is used together with methodology for use of contract oriented specifications. Such a study would address both the theses.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. Digial System Research Center. Research Report 66, October 1990.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Electrical/Computer Science and Engeneering Seris. Van Nostrand Reinhold Company, 1984.

[4] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. FOCUS on Streams, Interface, and Refinement*. Springer-Verlag, 2001.

[5] A. Cau and P. Collette. A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33:153–176, 1996.

[6] Folker den Braber. A precise foundation for modular refinement of MSC specifications. Master's thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2001.

[7] Anders Ek. Verifying Message Sequence Charts with the SDT Validator. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects*, pages 237–249. Elsevier, 1993.

[8] Norman Fenton and Shari Lawrence Pfleeger. Can formal methods always deliver? *IEEE Computer*, 30(2):34–34, February 1997.

[9] Kate Finney. Mathematical notation in formal specifications: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[10] Kate Finney and Norman Fenton. Evaluating the effectiveness of Z: The claims made about CICS and where we go from here. *J. Systems Software*, 35(3):209–216, 1996.

[11] Ursula Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*. PhD thesis, Fakultät für Informatik, Technischen Universität München, 1998.

[12] Echhardt Holz and Ketil Stølen. An attempt to embed a restricted version of SDL as a target language in Focus. In D. Hogrefe and S. Leue, editors, *Proc. Formal Description Techniques VII (FORTE'94)*, pages 324–339. Chapman and Hall, 1995.

[13] ITU-T. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120*, 2000.

[14] ITU-T. *Specification and description language (SDL), ITU-T, Recommendation Z.100*, 2000.

[15] Cliff B. Jones. *Development Methods for Computer Programs. Including a Notion of Interference*. PhD thesis, Oxford University, 1981.

[16] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt. Autolink. A tool for automatic test generation from SDL specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT98)*, 1998.

[17] Mass Soldal Lund. Validation of contract decomposition by testing. Master's thesis, Department of Informatics, University of Oslo, February 2002.

[18] Jayadev Misra and K. Mani Chandy. Proof of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–425, July 1981.

[19] Ketil Stølen. Assumption/commitment rules for dataflow networks — with an emphasis on completeness. In H. Riis Nielson, editor, *Proc. 6th European Symposium on Programming (ESOP'96)*, number 1058 in Lecture Notes in Computer Science, pages 356–372. Springer-Verlag, 1996.

[20] Telelogic. *Telelogic Tau 4.2. User's Manual*, 2001.