Testing Decomposition of Component Specifications Based on a Rule for Formal Verification

Mass Soldal Lund
Department of Informatics, University of Oslo, Norway
SINTEF Telecom and Informatics, Norway
mass.s.lund@sintef.no

Abstract

This paper proposes a general technique for testing decomposition of component specifications based on rules for formal verification. Component specifications are expressed as pairs of two models: a context assumption and a component guarantee. Thereby they capture the contract-like nature between the component to be developed and the context in which it is supposed to work. The paper provides empirical evidence that A/G rules developed for formal methods are highly relevant as patterns for testing decomposition of component specifications. More explicitly, that testing the validity of decomposition based on A/G rules is valid, and moreover, that testing based on A/G rules is more efficient that a conventional approach for the same kind of test-quality.

1. Introduction

Today's computerized systems are large and complex; they run on all kinds of platforms, and they are becoming an integrated part of infrastructure and of the society itself. Today's systems are distributed, which means that they consist of components distributed in space, and open in the sense that they run in environments that potentially can give the systems any input. The development of such systems is at least as complex as the systems themselves. At the same time there is demand for efficient and cost effective system development. In the development of software systems there is clearly a need for efficient methods and tools for ensuring the required quality.

Currently, model-driven and platform independent system development is emerging as a way of coping with these challenges, advocated by the Object Management Group (OMG) [3] – the main standardization body within software development, among others. Consider Figure 1. Model-driven system development implies that software systems are specified in a platform independent manner, for example using the Unified Modeling Language (UML) [23]. When a system is realized, the specification is "projected" through a specific platform "technology filter" (e.g., CORBA, .Net or J2EE). Then, by changing the "technology filter", the specification may be realized in different instances based on different platforms or combination of platforms. UML may be used for specifying both platform independent models and platform specific models [25].

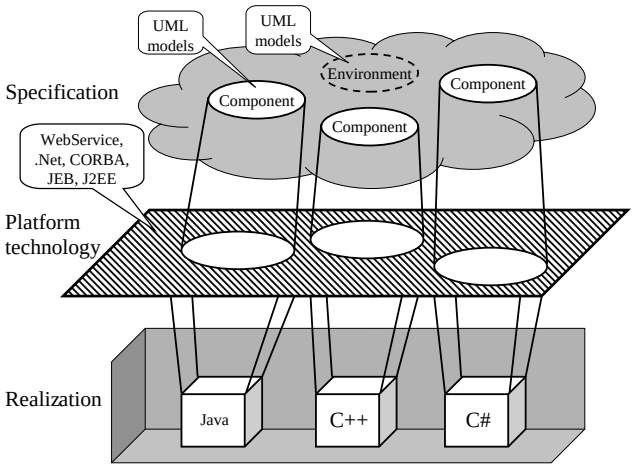

Figure 1. Platform independent system development

Testing is the standard method used for ensuring and increasing the quality and reliability of computerized system. Even though the use of models is increasing, testing has not yet found its place in model-driven development.

Testing has traditionally been carried out at the end of a system development process. Experience shows that a large amount of the time and money spent in system development is used for testing and error correction (estimates vary between 25% and 75%; see, e.g., [8, 24, 30]). The common practice is to do testing of integration after the components

are implemented. If it becomes evident that the components are not able to communicate or that the integration does not give the expected result, it may be necessary to make changes in the implementation, or even make new implementations, of some of the components. The cost of correcting an error increases the longer it takes before the error is discovered (see, e.g., [28]).

This paper proposes a general technique for testing decomposition of component specifications based on a rule for formal verification. Components are normally not expected to function in arbitrary contexts, but only in contexts that satisfy certain syntactic and semantic conditions. In order to make full use of a component description, these context conditions must be formalized. We therefore consider a component specification as a pair of two models:

- A model describing the relevant syntactic and semantic assumptions made about the component's context. In the following this model is referred to as the *context assumption*.
- A model describing the requirements to the component under the assumption that the component is executed in a context satisfying the context assumption. In the following this model is referred to as the *component guarantee*.

Specifications of this kind are a bit like contracts: the system developer commits to develop a component satisfying the component guarantee given that the context in which the component is supposed to run is correctly formalized by the context assumption. Contract-oriented specification has been suggested in many contexts and under different names. Within the RM-ODP [14] community one speaks of contracts related to quality of service specification [7]. In the formal methods community there are numerous variations; the pre/post [11], the rely/guarantee [16] and the assumption/guarantee [2] styles are all instances of contract-oriented specification. Other less formal examples are the design-by-contract paradigm, introduced by Bertrand Meyer [20], and the UML based approach advocated by Christine Mingins and Yu Liu [21].

In the following we use (A, G) to denote a component specification with context assumption A and component guarantee G. Testing decomposition of a component specification then means being able to test that the composite specification $(A_1, G_1)\|(A_2, G_2)$ of two interacting components[1] is a refinement of the overall specification (A, G). One approach is to test this directly by generating a representative set of test cases from the composite specification and check whether they are all allowed by the overall specification (A, G). We refer to this as the *conventional approach*. In the case of formal verification this corresponds to checking that the logical formula captured by the composite specification implies the overall specification. This is non-trivial because two component specifications (the one being the context of the other) may be mutually dependent, leading to circularities in the reasoning. This has motivated the development of specific rules (in the sequel referred to as *A/G rules*) for formal verification of contract decomposition, allowing the dependencies to be discharged in a structured manner. In [2, 16, 22, 26] such rules are presented with respect to TLA [18], the Owicki/Gries-language, CSP [12] and Focus [5], respectively. As argued in [6] these rules are all instances of a more general proof-principle, which we refer to as the *composition principle*. For example, as explained in [26], the Focus rule is basically the same as the TLA rule with the exception that the Focus rule handles assumptions with arbitrary liveness properties.

This paper provides evidence that A/G rules are also highly relevant patterns for organizing testing of contract decomposition. More explicitly, that testing the validity of decomposition based on A/G rules is valid, and moreover, that testing based on A/G rules is more efficient that the conventional approach for the same kind of test-quality.

In order to provide empirical evidence, two prototype tools have been developed; one based on an A/G rule, and one for the conventional approach. The context assumptions and component guarantees are expressed in the Specification and Description Language (SDL) [15]. SDL was selected mainly because of the availability of software for testing; we could just as well have used UML state machines or Java programs. We could also have restricted the use of SDL to the overall specification and expressed the composite specification in Java; the principles would have been just the same. Because we selected SDL, we decided to use the A/G rule in Focus since the relationship between SDL and Focus is well understood [4, 5, 9, 10, 13].

The remainder of this paper is structured as follows: Section 2 provides an SDL semantics for component specifications whose context assumptions and component guarantees are expressed in SDL. Furthermore, the relationship between the SDL semantics and component specifications expressed in Focus is established. With this mapping between SDL and Focus, analytic results formulated in the Focus semantics may be transferred to SDL specifications.

Section 3 describes and motivates the design of the two testing tools; the tool for the conventional approach and the tool based on the A/G rule in Focus. Section 4 summarizes and discusses the empirical results. Section 5 provides a brief summary and draws the main conclusions.

[1] In this paper we only consider composition of two components. This is however no restriction of the proposed strategy, which handle composition of an arbitrary number of components

2. A/G Specifications in SDL

An A/G specification consists of a context assumption A and a component guarantee G. The interpretation of an A/G specification (A, G) is that the component should fulfill its guarantee G *at least as long* as its environment fulfills the assumption A.

When composing A/G specifications there is a possibility of getting circularities when reasoning about the overall behavior of the composition. For example; in the composition $(A_1, G_1) \| (A_2, G_2)$ we might get the situation that G_2 fulfills A_1 if and only if G_1 fulfills A_2. The requirement that the guarantee must hold at least as long as the assumption prevents us from getting into this kind of circularities by providing a means for inductive reasoning [26]. The composition principle itself is based on induction and depends heavily on this property.

In Focus, which semantics is based on streams of messages with a discrete representation of time, "at least as long" is interpreted as "at least one time unit longer". In the case that the assumption is broken by the environment, the component may start behaving unpredictably one time unit later. This random behavior is a result of not specifying how the component will behave if the assumption is broken; we *do not know* how the component will behave.[2] Within formal methods this is a common technique for handling underspecification (see, e.g., [2, 17]).

Underspecification is common practice in all system development, because all decisions concerning a system are not made at the same time. For example while working on specifying the core functionality of a component, you do not want to bother yourself with platform dependent error and exception handling. Decisions regarding this are postponed until later, when correct operation of the component is specified.

2.1. SDL Semantics for Component Specifications

When specifying a component with SDL, a system developer will normally make implicit use of the A/G paradigm. He or she will write one specification of the component to be developed, and one specification capturing component relevant behavior of its environment. The latter is, for example, often required as a basis for testing. For simplicity, we assume in this paper that these two specifications are formalized as SDL processes.

An A/G specification of a component specified with SDL consists of two SDL processes; a process A specifying the context assumption and a process G specifying the component guarantee. This means that A specifies the relevant aspects of the environment of the component as seen from the component's point of view, while G specifies the behavior of the component the same way as a component specification usually would do.

In order to capture the requirements of A/G specifications explained above, the SDL processes A and G are composed into a special SDL block AG, shown in Figure 2. This block diagram provides an SDL representation of the "at least as long" interpretation outlined above. Note that this block diagram is created schematically from the two specifications A and G written by the system developer. We need this SDL characterization of A/G semantics for two purposes: (1) in order to apply results from Focus on SDL specifications; (2) as a basis for testing.

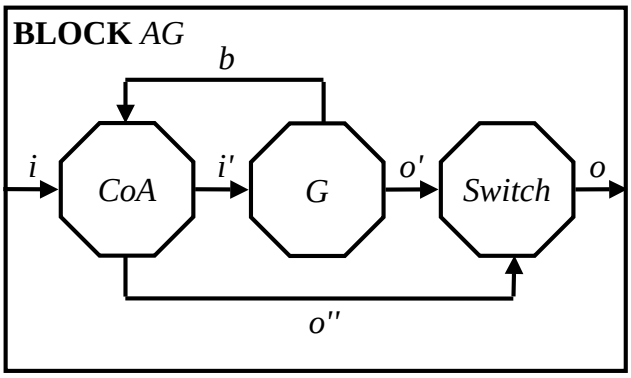

Figure 2. "At least as long" characterization in SDL

The input channel i represents the external environment, while the channel b is used for the component's feedback to the environment. o is the output channel. In this SDL block, A is schematically translated into a process CoA that instead of simulating the environment recognizes legal (i.e., assumed) input from the channels i and b and forwards it to G.[3] If ever illegal input is received, CoA will stop forwarding messages to G and instead start sending random output to the channel o''. As argued earlier, this is because the component is underspecified for this case. The $Switch$ process is initially forwarding messages only from o'. Upon receiving input on o'' the process $Switch$ will continue forwarding messages from o' to o for one unit of time and then switch to forwarding input only from o''.

In summary, the AG block simulates A/G specifications and the surroundings of the block act as an unpredictable environment.

[2] Notice that one way of behaving randomly is to behave correctly

[3] This solution was chosen because the input channel from the boundary simplifies monitoring of input and composition of AG blocks; the choice is solely pragmatic

2.2. Mapping the SDL Semantics to Focus

In this work we have only made use of a subset of SDL. This subset excludes most of the advanced features, and what remains are basic finite state machines and dataflow diagrams.[4] The mapping between SDL and Focus is well understood. Based on [9, 10, 13] we define semantics for SDL by translating SDL specifications into Focus specifications. In this translation, SDL specifications are expressed as predicates over infinite input and output streams.

Considering the schema presented in Figure 2, it intuitively makes sense to assume that CoA and $Switch$ are instantaneous; that is, they use no noticeable time in processing messages. Further it makes sense that there in G *will* be some delay since G specifies the behavior of a component.

In [19] we prove that if these assumptions are true, then $[\![AG]\!] = [\![(A, G)]\!]$.[5] In other words does the SDL block AG specify the same behavior as the Focus specification (A, G). However, the tool support we used as basis for our prototype implementations (see Section 3) was not able to handle these constraints on processing time. Instead we used the assumption that all SDL processes CoA, G and $Switch$ have time delay. In [19] we prove that we then have $[\![AG]\!] \subseteq [\![(A, G)]\!]$, i.e., that the AG block does not capture all the input/output histories of (A, G), but does not introduce new ones. This ensures that the soundness of the A/G rule formulated and proved in Focus carries over to the SDL semantics. Specifying and simulating components with the AG block schema do not introduce new behavior that could violate the verification strategy. If an AG block specifies histories that for some reasons should not be possible, these histories are also specified by the corresponding Focus specification, and the A/G rule will be able to catch them.

3. Testing Tools

In this section we describe two testing strategies named the *conventional approach* and the *A/G approach*, and how these were implemented as two prototype tools. Both strategies do testing of behavioral refinement. Behavioral refinement means adding detail on how the specified system or component should behave. Since decomposition is a kind of behavioral refinement, testing of behavioral refinement can be applied for testing decomposition.

A specification $Spec_1$ is refined by a specification $Spec_2$, denoted $Spec_1 \rightsquigarrow Spec_2$, iff $[\![Spec_2]\!] \subseteq [\![Spec_1]\!]$, i.e., the behavior (the set of possible input/output histories) specified by $Spec_2$ is a subset of the behavior specified by $Spec_1$ [2, 5, 17]. In this setting, $Spec_2$ is a decomposition of $Spec_1$.

Testing of the refinement relation is done by generating test cases from $Spec_2$ and executing them against $Spec_1$, as illustrated in Figure 3. The tools do black box testing (also called functional testing) [29], and hence are the test cases recordings only of external input and output.

If all test cases generated from $Spec_2$ are executed as possible histories of $Spec_1$, this indicates that the refinement is valid since the subset relation holds. On the other hand, if one or more of the test cases are not executable by $Spec_1$, we know that the refinement is invalid because the subset relation is broken.

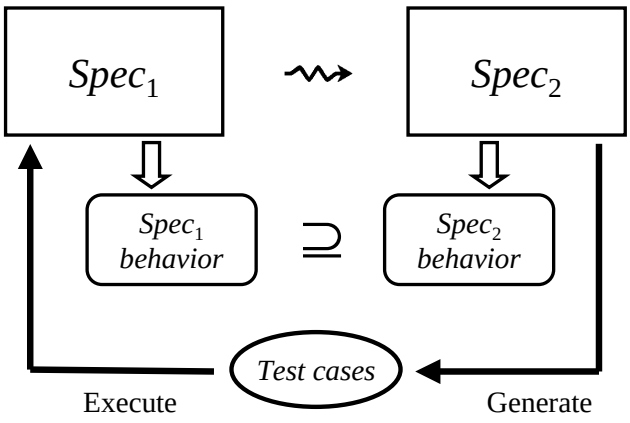

Figure 3. Testing of refinement

The testing tools are based on SDL Validator, which is a part of the SDL tool Telelogic Tau [27]. In the following subsections the two approaches are presented briefly. Details on the implementation are found in [19].

3.1. Tool for Conventional Approach

The conventional approach is based on performing tests in the most intuitive way, namely generating test cases from the network of sub-component specifications and executing them against the specification that was decomposed. This strategy is illustrated in Figure 4 (without showing connections between sub-components). Each of the rectangular boxes in this figure represents an AG block as shown in Figure 2. In other words, (A, G), (A_1, G_1) and (A_2, G_2) are specified by the system developer in accordance with the block diagram in Figure 2. The interaction between the two blocks on the left is based on the semantics of standard parallel composition in SDL.

If one or more of the test cases generated from the specification on the left side of the figure are not executable by the specification on the right side, the decomposition is not valid. Therefore, if all test cases are executable there is a strong indication that the decomposition is valid.

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

[5] The denotation $[\![Spec]\!]$ of a specifi cation $Spec$ is the set of all possible input/output histories of $Spec$

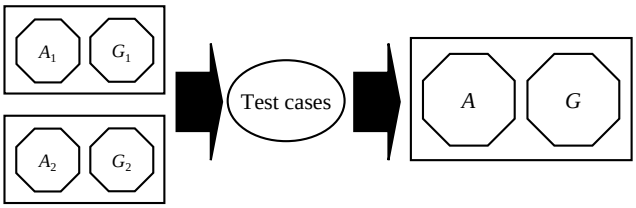

Figure 4. Conventional approach

3.2. Tool for A/G Approach

The A/G approach is based on the composition principle, or more specifically, a simplification of an instance (A/G rule) formulated for Focus in [26]. Applying this rule leads to a setup for testing as shown in Figure 5. The SDL processes on the left side of the figure interact based on the semantics of standard SDL parallel composition, and *not* the special A/G semantics of the SDL block in Figure 2. As argued in [19], the validity of the approach follows from the theorems described in Section 2.2.

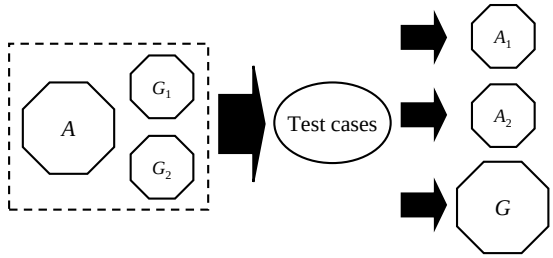

Figure 5. A/G approach

Each of the arrows on the right side of the test cases in the figure represents an execution of the set of test cases. This means that the set of generated test cases is executed three times when the decomposition consists of two sub-components. Each of these executions represents one premise of the A/G rule. If one or more generated test cases are not executable in one or more of these executions, the decomposition is not valid. This means that if all test cases are executable in all executions this is a strong indication that the decomposition is valid.

4. Empirical Results

In order to validate and compare the two testing strategies, 15 example decompositions, in the sequel referred to as *decompositions*, were tested by both tools. The decompositions cover a range of situations that may occur when A/G specifications are decomposed and are representative for a considerable class of decompositions, even though they are artificial examples. For simplicity, environment assumptions expressed in terms of component behavior were not present in any of the decompositions since the A/G rule provides the same kind of structured induction to eliminate mutual dependencies for the general case as for the decompositions we used in our study. With respect to Figure 2, this means that none of the assumptions were dependent on the feedback channel b.

Our empirical results are presented in Table 1. The table is divided into four parts. The first part shows the number of states in the SDL specifications of the decompositions, as a measure of size. The second part ("Analysis") shows the results from manually analyzing the validity of the decompositions. When working with A/G specifications, there are two ways of making valid refinements [1, 5]: (1) strengthening the guarantee; (2) weakening the assumption.

An assumption or guarantee P is strengthened if the refinement P' implies P, i.e., $[\![P']\!] \Rightarrow [\![P]\!]$, and weakened if $[\![P]\!] \Rightarrow [\![P']\!]$. In the context of sets of histories, implication is interpreted as subset, so P is strengthened if $[\![P']\!] \subseteq [\![P]\!]$ and weakened if $[\![P]\!] \subseteq [\![P']\!]$. As mentioned earlier, we are here only concerned about external behavior.

Hence, a decomposition is analyzed to be invalid iff the assumption is strengthened (S), the guarantee is weakened (W), or both. In this analysis, a network of sub-components (which is the result of a decomposition) is viewed as a black box, i.e., we are concerned about the overall assumptions and guarantees.

The two last parts of the table ("Conventional" and "A/G") show the results of testing the decompositions with the two tools. In addition, some measurements for comparing the tools are provided: execution time, the number of test cases generated, the average length (number of messages) of these test cases and finally the symbol coverage (the percentage of SDL symbols visited).

From the table we see that both tools gave the same results with respect to verification. Further, these results were the same as the results from analyzing the decompositions manually, so the tools verified exactly the valid decompositions. In the following subsection we analyze the results presented in the table with respect to effectiveness.

4.1. Efficiency

If we look at measured execution time, number of test cases and average length of test cases in the table, we see that the seven decompositions 1, 2, 3, 4, 10, 14 and 15 stand out with the following characteristics:

- The tool using the conventional approach used considerably longer time that the tool using the A/G approach.
- The tool using the conventional approach generated

Table 1. Efficiency of testing tools

	Decomposition	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	Number of states	7	15	20	10	8	6	6	7	8	14	6	7	9	9	7
Analysis	Assumption	-	-	-	S	-	-	-	W	W	-	-	-	-	-	S
	Guarantee	-	-	-	-	S	S	W	-	W	W	W	-	S	S	-
	Valid	Y	Y	Y	N	Y	Y	N	Y	N	N	N	Y	Y	Y	N
Conventional	Verified	Y	Y	Y	N	Y	Y	N	Y	N	N	N	Y	Y	Y	N
	Execution time	7:25	6:33	11:37	10:08	0:04	0:09	0:08	0:04	0:04	7:06	0:38	0:22	0:15	0:48	6:57
	Test cases	10	15	14	10	4	3	3	2	2	10	4	1	1	5	10
	Average length	94.4	1.5	121.2	74.8	3.0	1.7	2.0	2.0	3.0	117.9	2.0	1.0	20.0	41.4	116.4
	Coverage	95.2	70.0	96.1	97.6	100.0	83.4	100.0	61.8	61.8	88.7	100.0	43.8	43.8	96.4	100.0
A/G	Verified	Y	Y	Y	N	Y	Y	N	Y	N	N	N	Y	Y	Y	N
	Execution time	0:11	0:16	0:20	0:14	0:14	0:15	0:16	0:12	0:13	0:13	0:48	2:49	0:28	0:12	0:12
	Test cases	2	6	5	4	4	3	3	2	2	2	5	5	1	2	2
	Average length	4.0	4.3	5.6	2.0	3.0	3.7	4.0	6.0	3.0	8.0	84.0	158.0	143.0	3.0	4.0
	Coverage	100.0	100.0	97.9	100.0	100.0	100.0	100.0	100.0	89.3	100.0	100.0	82.9	82.9	100.0	100.0

more test cases than the tool using the A/G approach.

- The tool using the conventional approach generated considerably longer test cases than the tool using the A/G approach (except for Decomposition 2).

These decompositions also have another important characteristic that the other eight do not have: The resulting network of sub-components have assumptions that are non-trivial. By a trivial assumption we mean an assumption that does not put any restrictions on the assumed input to the specified component, and hence is superfluous.

The case of having non-trivial assumptions is the normal case; if all assumptions are trivial, then we are no longer working with A/G specifications and lose the advantage of considering the context of components. Also, if all assumptions are trivial the two testing strategies are in principle equal, so we would not expect any big differences in the efficiency.

We draw the conclusion that for the important case where the decomposition has non-trivial assumptions, the A/G approach is considerably more efficient than the conventional approach. We also observe that the A/G approach gave better coverage than the conventional approach, and hence the gained efficiency is not a result of incomplete tests. The difference occurs because the conventional approach produces a large amount of random behavior when assumptions are broken. The A/G rule is used to eliminate this random behavior in a mathematical sound manner.

5. Conclusions

This paper has proposed a general technique for testing decomposition of component specifications expressed as pairs of two models: (1) a context assumption describing the surroundings in which the component is supposed to work; (2) a component guarantee describing the requirements to the component with respect to the context assumption. The paper has provided evidence that A/G rules are highly relevant patterns for organizing testing of contract decomposition. More explicitly, that testing the validity of decomposition based on A/G rules is valid, and moreover, that testing based on A/G rules is more efficient than the conventional approach for the same kind of test-quality. The validity has been proved with respect to a semantic mapping of SDL to Focus [9, 10, 13] and the A/G rule in [26]. The empirical evidence results from testing of 15 decomposition. Although the decompositions in the study were small and artificial, they were carefully designed to be representative for a considerable class of decompositions.

Acknowledgement

The work on this paper was partially funded by the Research Council of Norway Basic ICT Research program project SARDAS (15295/431).

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Prog. Lang. Syst.*, 15(1):73–132, Jan. 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–534, May 1995.

[3] Architecture Board ORMSC. *Model Driven Architecture (MDA). Document number ormsc/2001-07-01*. Object Management Group, 2001.

[4] M. Broy. Towards a formal foundation of the Specification and Description Language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[5] M. Broy and K. Stølen. *Specification and development of interactive systems. FOCUS on streams, interface, and refinement*. Springer-Verlag, 2001.

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

[7] A. Février, E. Najm, and J. B. Stefani. Contracts for ODP. In M. Bertran and T. Rus, editors, *Proc. Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software (ARTS'97)*, number 1231 in Lecture Notes in Computer Science, pages 216–232. Springer-Verlag, 1997.

[8] B. Hailpern and P. Santhanam. Software debugging, testing and verification. *IBM Syst. J.*, 41(1):4–12, 2002.

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

[10] U. Hinkel. Verification of SDL specifications on base of a stream semantics. In Y. Lahar, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. 1st workshop of the SDL Forum Society on SDL and MSC, Volume II*, pages 241–250, 1998.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, Oct. 1969.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Series in computer science. Prentice-Hall, 1985.

[13] E. Holz and K. 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.

[14] ISO/IEC 10746. *Basic reference model for open distributed processing*, 1995.

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

[16] C. B. Jones. *Development methods for computer programs. Including a notion of interference*. PhD thesis, Oxford University, 1981.

[17] C. B. Jones. *Systematic software development using VDM*. Series in computer science. Prentice-Hall, second edition, 1990.

[18] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[19] M. S. Lund. Validation of contract decomposition by testing. Master's thesis, Department of Informatics, University of Oslo, Feb. 2002.

[20] B. Meyer. *Object-oriented software construction*. Prentice-Hall, 1997.

[21] C. Mingins and Y. Liu. From UML to design by contract. *Journal of Object-Oriented Programming*, pages 6–9, Apr. 2001.

[22] J. Misra and K. M. Chandy. Proof of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–425, July 1981.

[23] OMG-UML. *Unified Modeling Language Specification, version 1.4*. Object Management Group, 2001.

[24] M. Roper. Software testing—searching for the missing link. *Information and Software Technology*, 14:991–994, 1999.

[25] R. K. Runde and K. Stølen. What is model driven architecture? Research Report 304, Department of Informatics, University of Oslo, Mar. 2003.

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

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

[28] J. C. Westland. The cost of errors in software development: evidence from industry. *J. Systems Software*, 62:1–9, 2002.

[29] J. A. Whittaker. What is software testing? And why is it so hard? *IEEE Software*, 17(1):70–79, 2000.

[30] B. Woodworth. Message from the Corporate Director, IBM Software Test. *IBM Syst. J.*, 41(1), 2002.