Reasoning About Compositions of Concerns

Marsha Chechik **Steve Easterbrook**
Department of Computer Science
University of Toronto
6, King's College Rd
Toronto, Canada M5S 3H5
+1 416 978 3610
{chechik,sme}@cs.toronto.edu

Abstract
In this paper we propose a framework for defining and reasoning about compositions of concerns, based on multi-valued logics.

1 Introduction

Separation of concerns refers to the ability to identify, encapsulate and manipulate parts of software that are relevant to a particular concept, goal, task or purpose [23]. For example, sources of individual concerns can come from software engineers playing different roles in software development (e.g., UI builders and database builders), or from stakeholders with different views on the system (e.g., end users and performance engineers). An effective way to specify individual concerns is via viewpoints-based approaches [3, 17]. These approaches allow to gather and maintain (possibly inconsistent and incomplete) information gathered from multiple sources. They explicitly separate the descriptions provided by different stakeholders, and concentrate on identifying and resolving conflicts between them.

Clearly, many concerns overlap and interact. To make the concept of separation of concerns truly general and usable, we need to reason about properties of the composed system, that is, reason about *compositions* of concerns. We might like to know whether all concerns from which the software is composed are *mutually consistent*, but an even more important problem is to determine whether the composed system is *correct*, that is, whether it satisfies certain agreed-upon properties. It is often impossible to assess properties of the composed system based only on descriptions of individual concerns. For example, if we specify a telephone system by describing individual concerns of callees and callers, then we have to compose the concerns before determining whether the billing for the caller starts only after the callee picks up.

Concerns may need to be composed in a variety of ways, depending in the roles they play in the overall system description. For example, concerns may represent parallel devices, different views of several users specifying their concerns of the system (e.g., a bank manager and a teller), different levels of abstraction [19], different descriptions of the complete system (e.g., from the point of view of performance analysis and that of data management), etc. The goal of this paper is to address the problem of composing descriptions of overlapping concerns and reasoning about property preservation in the compositions.

Compositions are typically embedded in the specification language. The reason for this embedding is that many compositions are very domain-specific. Clearly, domain-specific reasoning is essential, and we have gone a long way with it, with many extremely successful examples. However, effective reasoning in domain-specific languages requires creation of high quality analysis tools. Unfortunately, the engineering that goes into these tools needs to be reproduced for each individual domain-specific language. A more serious problem is that concerns in a large system almost always come from *different* domains, thus making domain-specific languages inapplicable.

As an alternative approach, category theory has been proposed as framework for adding compositionality to existing specification languages [18]. It has been successfully used for both algebraic [15] and temporal logic [16] specification languages. Category theory provides a rich body of theory for reasoning about structured sets of objects (in this case, specifications and their interconnections), without embedding this structure into the language. However, existing applications of category theory have provided only one type of specification interconnection, that of *refinement*. While some other types of composition can be expressed in terms of structures of refinement relations [15], this restriction has limited the application of category theory to program generation by refinement. The abstractness of category theory has also limited its appeal for practical software engineering.

Finally, if we leave the world of domain-specific languages, we find ourselves in the situation where we cannot predict all possible compositions that may be useful in practice. Attempts to do so typically result in dozens of operators, making them too specialized and too hard to understand and use. Instead, we propose a mechanism to define *arbitrary* compositions, that is, lift compositions to the first-class status in our specification/verification framework. Some compositions are "built-in", and there are provisions for defining new compositions. The need to specify different compositions has also been recognized by the developers of the language Picola [20, 2]. This language is designed to allow compositions between objects in OO programs. Our approach is complementary, aimed at reasoning about possibly inconsistent requirements specifications of the different concerns.

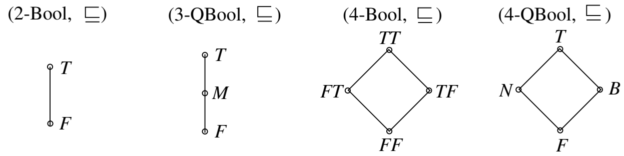
The choice of compositions is determined by the task at hand and by the desired level of abstraction. Suppose we are given several individual descriptions of concerns. We might want to preserve information about how many descriptions gave each type of response to a given question (i.e., so '4 trues, 1 false' is distinct from '3 trues, 2 falses'), rather than just concluding that an inconsistency is present. We might also preserve information about the exact source of each piece of information (i.e. so that 'A said true, B said false' is distinct from 'A said false, B said true'). Alternatively, we might allow individual descriptions of concerns to indicate that they 'do not know' answers to some questions, or designate some descriptions as authorities, so that their answers carry more weight in the final composition.

With this goal in mind, we propose the use of *paraconsistent logics* as a means of specifying and reasoning about a variety of compositions. For example, a four-valued logic with values *True*, *False*, *Both* and *Neither* has been first used by Belnap [4] in 1977 in the context of reasoning about inconsistent updates to a database. Here the system represents one concern, and each individual update represents a different concern. Compositions of potentially inconsistent concerns can be effectively described using *abduction* [21, 22]. Abduction creates a number of consistent worlds, where each world represents a possible resolution of inconsistency. Unfortunately, abductive reasoning is basically a limited version of *composition as conjunction* [1, 24], and cannot support other compositions.

Instead, we argue that a family of non-classical logics, called *quasi-boolean multi-valued logics* [5] allow a natural way to specify and reason about not only all classical compositions, e.g. conjunction, disjunction, assume-guarantee, but also many other interesting and potentially useful compositions. The rest of this paper is organized as follows: in Section 2 we describe quasi-boolean multi-valued logics and give examples of their use on several compositions. In Section 3 we briefly describe the work we have done on proving support for automated verification of the composed system w.r.t. global correctness properties expressed in CTL – a branching-time temporal logic. We also address possible interpretations of verification results. We conclude in Section 4 with a brief discussion of the overall composition/verification framework.

2 Multi-Valued Logics

Multi-valued logics use additional truth values to represent levels of contradiction or uncertainly. For example, a three-valued logic has an additional value *Maybe* to represent uncertainty, and has been used by a number of researchers in the context of reasoning about abstractions (cf. [6, 13]). Given a finite number of truth values, a logic can use tables to define conjunction, disjunction and negation operators on these values. However, if we wish to keep useful properties of the classical logic, such as associativity, idempotency, distributivity, and De Morgan's laws, we can define a logic using a lattice of truth values, with *True* at the top and *False* at the bottom. Then conjunction and disjunction defined as meet and join operations on elements of the lattice are commutative, associative, and idempotent. Some examples of lattices appear in Figure 1. For example, for the logic (4-Bool, $\sqsubseteq$), $FT \vee TF = FT \sqcup TF = TT$, and $FT \wedge TF = FT \sqcap TF = FF$. $a \sqsubseteq b$ means that a is "less true than" b.



a) Classical logic b) Representing uncertainty c) Representing disagreement d) Belnap's 4-valued logic

Figure 1: Four quasi-boolean logics.

A logic is called *quasi-boolean* [5] if it has a negation operator $\neg$ with the following properties (a, b are elements of the lattice):

$$\begin{array}{rcll} \neg(a \sqcap b) & = & \neg a \sqcup \neg b & \text{(de Morgan)} \\ \neg(a \sqcup b) & = & \neg a \sqcap \neg b & \\ \neg\neg a & = & a & (\neg \text{ involution}) \\ a \sqsubseteq b & \Leftrightarrow & \neg a \sqsupseteq \neg b & (\neg \text{ antimonotonic}) \end{array}$$

For example, we can define $\neg FT = TF$ and $\neg TF = FT$, or, for the three-valued logic, $\neg M = M$. More information about these logics is available in [11]. Such logics do not necessarily preserve the law of excluded middle ($a \vee \neg a = \top$) or a law of non-contradition ($a \wedge \neg a = \bot$), and thus allow reasoning in the presence of inconsistencies and incompletenesses. Note that classical boolean logic is included in this family as a 2-valued lattice — we refer to it as (2-Bool, $\sqsubseteq$).

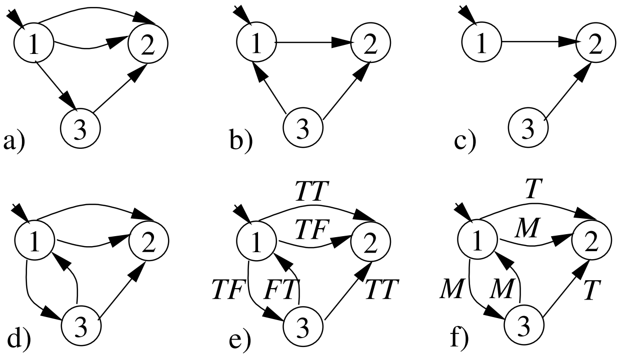


Figure 2: Two initial graphs (a)-(b) and their four compositions: (c) conjunction, (d) disjunction, (e) source preservation, (f) source abstraction.

We illustrate the role of multi-valued logics in specifying compositions using the example in Figure 2. Let Figure 2(a) and (b) represent views that Alice and Bob, respectively, hold about a graph. They agree on the states but disagree on transitions between them. The logics used by these viewpoints, L_A and L_B, respectively, are the same – (2-Bool, $\sqsubseteq$). In fact, usually logics used for modeling individual viewpoints are either classical or three-valued, if uncertainty is present. We can now combine Alice's and Bob's viewpoints; we start by defining L_C – the logic of the combined model. In Figure 2(c)-(d), L_C is the classical logic. Now, for each $A \in L_A$, $B \in L_B$, we map (A, B) to some element of L_C. This determines the composition. For example, *conjunction* is a mapping (2-Bool, $\sqsubseteq$) × (2-Bool, $\sqsubseteq$) → (2-Bool, $\sqsubseteq$) where (*T*, *T*) maps to *T*, and all other tuples map to *F*. The result of this merge is illustrated in Figure 2(c). *Disjunction* is a mapping where (*F*, *F*) maps to *F*, and all other tuples map to *T*. The result is given in Figure 2(d). Note that conjunction and disjunction abstract away the exact sources of information, which might be useful for further reasoning.

More interesting merges are also possible. For example, if we like to know which viewpoint contributed each edge in the merged model, we let L_C be (4-Bool, $\sqsubseteq$), with mappings (*T*, *T*) → *TT*, (*T*, *F*) → *TF*, etc. To denote these values on the resulting graph, we adopt the convention of labeling edges. Transitions labeled with *False*, (or *F* or *FF*) are not shown. Figure 2(e) gives an example of this type of composition. Note that in this composition the question "is there an edge (**1**,**3**) or edge (**3**,**1**) in both viewpoints" is answered positively (*TF* ∨ *FT* = *TT*).

Another useful composition might be the one where we want to note disagreement, but not necessarily identify its source. In that case, we can use the logic (3-QBool, $\sqsubseteq$), with mappings (*T*, *T*) → *T*, (*F*, *F*) → *F*, and everything else to *M*. Result of this merge is given in Figure 2(f). In this merge, the above question cannot be answered conclusively, since *M* ∨ *M* = *M*.

As mentioned earlier, the choice of the resulting logic is usually determined by the task at hand. Furthermore, we need rules that help us determine the meaning of the value of the property. For example, in our our last merge, the fact that the answer to "is there an edge (**1**,**3**) or edge (**3**,**1**) in both viewpoints" is answered with value *M* does not indicate that the viewpoints disagree. Three-valued logic compactly encodes disagreement, but only *T* and *F* values can be trusted. Many *M* answers actually do not present disagreement [7]. On the other hand, our mapping into (4-Bool, $\sqsubseteq$) is completely property-preserving: we can always correctly determine the interpretation of answers given by the merged model.

3 Multi-Valued Analysis

Although multi-valued logics provide a natural way for specifying compositions, we are mostly interested in analyzing resulting models w.r.t. preservation of desired properties. In addition, we are interested in being able to interpret results of this analysis correctly, since clearly the interpretation differs for the various types of composition performed. In this section we briefly describe the analysis we can perform if individual concerns are expressed as state machines. We describe the state machine models, the language for specifying properties and the multi-valued model-checking algorithm. This work is discussed in more detail in [14, 11, 8, 9].

In this work we assume that individual concerns are described as multi-valued finite-state machines (we call them χviews). Conventionally, a state machine model consists of a set of states, a set of transitions between states, and a set of variables whose values vary from state to state. We extend this model by associating each χview with a specific quasi-boolean logic. 'Boolean' variables now range over the values of the logic, rather than just being *True* or *False*. Transitions between states are also labeled with values from the logic. For example, in Figure 2(e), the value of (**1**,**3**) is *TF*. Figure 3 gives more realistic examples of χviews. Figure 3(a) depicts an aspect of a thermostat system that runs a heater when the temperature falls below desired. The system has one indicator (`Below`), a switch to turn it off and on (`Running`) and a variable indicating whether the heater is running (`Heat`). Figure 3(b) describes another aspect of the thermostat system – running the air conditioner. The behavior is similar, but this system handles the failure of the temperature indicator. If the temperature reading cannot be obtained in states `AC` or $\texttt{IDLE}_2$, the system transitions into state $\texttt{IDLE}_1$. Finally, Figure 3(c) contains a merged model, describing the behavior of the thermostat that can run both the heater and the air conditioner. The logic of these three models is (3-QBool, $\sqsubseteq$).

Since our χviews represent behavioural models, we would like to be able to ask them questions expressed in temporal logic. The language we chose for it is χCTL [11] – an extension of branching-time temporal logic CTL [12][1]. Properties expressed in CTL are evaluated on a tree of infinite computations produced from the finite-state machine. The syntax of χCTL is the same as that of CTL:

1. Every atomic proposition $a \in A$ is a CTL formula.
2. If φ and ψ are CTL formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $EX\varphi$, $AX\varphi$, $EF\varphi$, $AF\varphi$, $E[\varphi U \psi]$, $A[\varphi U \psi]$, $EG(\varphi)$, $AG(\varphi)$.

The temporal quantifiers have two components: A and E quantify over paths, while X, F, U and G indicate "next state", "eventually (*future*)", "until", and "always (*globally*)", respectively. Hence, $AX\varphi$ is *True* in state s if φ is *True* in the *next* state on *all paths* from s. $E[\varphi U \psi]$ is *True* in state *s* if *there exists* a path from s on which φ is *True* at every step *until* ψ becomes *True*. In χCTL, the semantics of quantification is different – "exists" is evaluated as a disjunction over paths, and "forall" is evaluated as a conjunction. For example, some properties of the thermostat system are:

1. Can the system transition into $\texttt{IDLE}_1$ from everywhere?
   $AG\ EX\ \texttt{IDLE}_1$
2. Is the heat on only if air conditioning is off?

[1] A multi-valued extension of LTL is also available and described in [10].

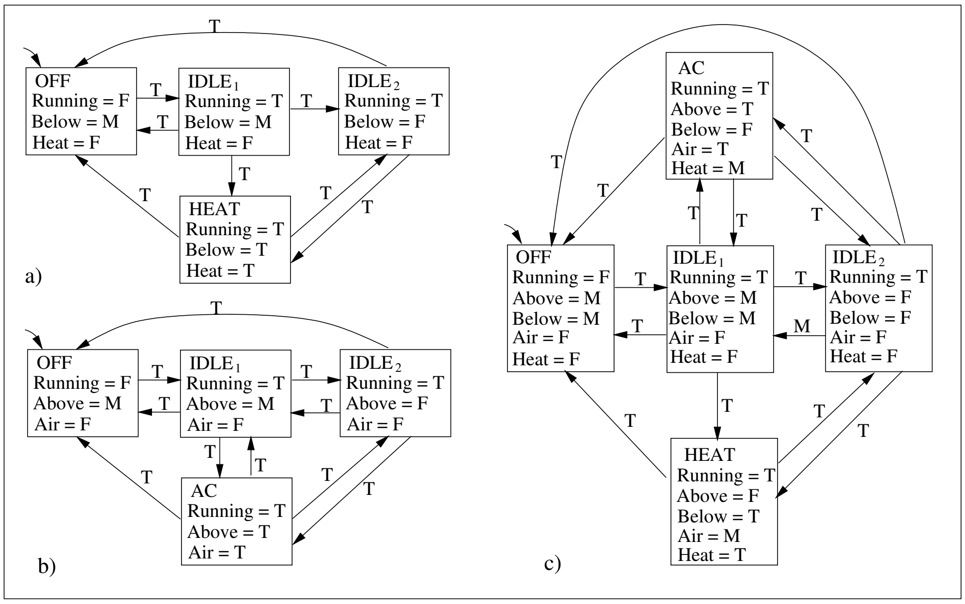


Figure 3: Models of the thermostat. (a) Heat only; (b) AC only; (c) combined model.

AG (Heat $\leftrightarrow$ Air)

3. Can heat be on when the temperature is above desired?
   AG (Above $\rightarrow \neg$Heat)

Note that the last two properties can only be expressed on a combined model. [11] gives the full semantics of χCTL.

We have developed a symbolic multi-valued χCTL model checker χchek [11, 8, 9] for analyzing χviews. The model checker takes as input a χview, including the definition of its quasi-boolean logic, and a temporal logic property expressed in χCTL, and returns a value from the logic representing the value that the χCTL property takes in the initial state of the χview, together with a counter-example, if applicable.

Once the answer is received, we would like to know its meaning w.r.t. individual descriptions of concerns. Suppose the model-checker returned *F* as the value of the first property of the thermostat. Does this mean that the property is *False* in both viewpoints? at least one viewpoint? cannot be concluded? To understand the meaning of some of the compositions, let us turn our attention back to the graphs in Figure 2. Under the "conjunction" composition (Figure 2(c)), we have a path in the composed graph only if this path is present in both of the initial graphs (that is why this composition is often called "conservative"). Thus, if an existential property is *True* in the composition, it is also *True* in each individual model. For example, "state **2** is reachable from the initial state", expressed in CTL as EF **2**, is *True* in the combined model and in the original models. However, if an existential property is *False* in the composition, in general no information is known about it in the original models. Universally-quantified properties have the opposite effect: *False*s are preserved, whereas *True*s are not. It is interesting to note that if both viewpoints agree on values of a property, it is not guaranteed that the property will have the same value in the combined model. For example, an existential property is *True* in a combined model only if individual viewpoints agree that it is *True* AND agree on the individual path that makes this property *True*. Otherwise no information about the value of the property in the combined model can be deduced.

The "disjunction" composition (Figure 2(d)) is often called "optimistic", because it includes all paths of the individual viewpoints (and possibly a few others). Thus, if a universally-quantified property is *True* on the combined model, it is *True* on each viewpoint. For example, "only states **2** and **3** are immediately reachable from **1**", formalized as AX (**2** $\vee$ **3**), is *True* in the graph in Figure 2(d), and also in the individual graphs. Further, if an existentially-quantified property is *False* in the combined model, it is *False* in each viewpoint.

The "abstraction" composition (Figure 2(f)) ends up preserving more properties than either "optimistic" or "pessimistic" compositions. If a property, either universal or existential, is *True* (*False*) in a combined system, it is *True* (*False*) on both individual systems. Consider, for example, a property "**2** is immediately reachable from **1**", expressed in CTL as EX **2**. This property is *False* in the combined model, and in graphs in Figure 2(a)-(b). However, no information is known when a property yields *M* on the combined model. Furthermore, the more concerns we compose, and the more interesting properties we want to verify, the more often will the answers be *M*. When most properties of interest are *M* in the combined model, then the abstraction is not effective, and other means of combining information should be explored. For example, we can go from a relatively compact 3-valued logic to more-valued logics, e.g. (4-Bool, $\sqsubseteq$), (8-Bool, $\sqsubseteq$), etc., which preserve the source of individual information and thus preserve more properties.

4 Discussion and Conclusion

In this paper we argued for the need to bring composition operators into first-class status for reasoning about concerns. Multi-valued logics can form the basis of such a language of compositions because they allow specification of all classical compositions and many other interesting compositions. Furthermore, over the past year we have developed an automated verification engine for checking multi-valued systems against temporal properties. This engine, χchek, is part of χbel[2] (Multi-Valued **B**elief **E**xploration **L**ogics) framework for merging and reasoning about individual concerns. The framework additionally includes support for merging namespaces of individual viewpoint descriptions, a few templates for the merge, and strategies for using the results of the analysis for negotiation.

Although χbel provides interpretations for several common compositions, more compositions will clearly be necessary. New compositions (merges) can be defined easily; however, the interpretation of property preservation is somewhat more difficult, and often requires considerable formal methods expertise. We are maintaining a library of useful compositions, and as our experience with the framework matures, the library will grow. The use of all compositions from the library is simple, and does not require formal methods expertise. The library will also include some domain-specific compositions, e.g., those occurring in the analysis of feature interactions in telecom, or reasoning about high assurance systems.

ACKNOWLEDGEMENTS

The ideas presented in this paper were formed in collaboration with members of the University of Toronto formal methods reading group. Special thanks to Arie Gurfinkel for carefully reading an earlier draft of this paper.

REFERENCES

[1] M. Abadi and L. Lamport. "Composing Specifications". *ACM Transactions on Programming Languages and Systems*, 15(1):73–131, January 1993.

[2] F. Achermann. "Language support for feature mixing". In *Proceedings of the Workshop on Multi-Dimantional Separation of Concerns in Software Engineering (at ICSE'00)*, June 2000.

[3] J. K. B. Nuseibeh and A. Finkelstein. "Framework for Expressing the Relatinoship Between Multiple Views in Requirements Specifications". *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[4] N. Belnap. "A Useful Four-Valued Logic". In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.

[5] L. Bolc and P. Borowik. *Many-Valued Logics*. Springer-Verlag, 1992.

[6] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 274–287, 1999.

[2] pronounced "Ki-bel"

[7] M. Chechik. "On Interpreting Results of Model-Checking with Abstraction". CSRG Technical Report 417, University of Toronto, Department of Computer Science, September 2000.

[8] M. Chechik, B. Devereux, and S. Easterbrook. "Implementing a Multi-Valued Symbolic Model-Checker". In *Proceedings of TACAS'01*, April 2001. To appear.

[9] M. Chechik, B. Devereux, S. Easterbrook, A. Lai, and V. Petrovykh. "Efficient Multiple-Valued Model-Checking Using Lattice Representations". Submitted for publication, January 2001.

[10] M. Chechik, B. Devereux, and A. Gurfinkel. "Model-Checking Infinite State-Space Systems with Fine-Grained Abstractions Using SPIN". In *Proceedings of the 8th SPIN Workshop*, May 2001. To appear.

[11] M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proceedings of FME'01*, pages 72–98, March 2001.

[12] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 464–475, June 1989.

[13] D. Dams, R. Gerth, and O. Grumberg. "Fair Model Checking of Abstractions". In *Proceedings of VCL00*, 2000.

[14] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, May 2001.

[15] H. Ehrig and B. Mahr. *"Fundamentals of Algebraic Specification 2"*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990. Modules specifications and constraints.

[16] J. Fiadeiro and T. Maibaum. "Temporal Theories as Modularisation Units for Concurrent System Specification". *Formal Aspects of Computing*, 3(4):239–272, 1992.

[17] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. "Inconsistency Handling in Multi-Perspective Specifications". *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[18] J. Goguen. "Reusing and Interconnecting Software Components". *IEEE Computer*, 19(2), 1986.

[19] M. Jackson. *Problem Frames. Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[20] M. Lumpe, F. Achermann, and O. Nierstrasz. "A Formal Language for Composition". In *Foundations of Component Based Systems*, pages 69–90, 2000.

[21] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. "An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering". In *Proceedings of the Fourth International Symposium on Requirements Engineering (RE'99)*, Limerick, Ireland, June 7-11 1999. IEEE Computer Society Press.

[22] A. Russo, R. Miller, B. Nuseibeh, and J. Kramer. "An Abductive Approach for Handling Inconsistencies in SCR Specifications". In *Proceedings of ICSE2000 Workshop on Intelligent Software Engineering*, 2000.

[23] P. Tarr and H. Ossher. Workshop on advanced separation of concerns in software engineering at icse 2001. Call for Papers, 2001.

[24] P. Zave and M. Jackson. "Conjunction as Composition". *ACM Transactions on Sofware Engineering and Methodology*, 4(2):379–411, 1993.