The Burden of Modularity

Introduction to ISoLA 2020 Track on *Modularity and (De-)composition in Verification*

Dilian Gurov[1], Reiner Hähnle[2], and Eduard Kamburjan[2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
dilian@kth.se
[2] Technische Universität Darmstadt, Darmstadt, Germany
{reiner.haehnle,eduard.kamburjan}@tu-darmstadt.de

Abstract. Modularity and compositionality in verification frameworks occur within different contexts: the model that is the verification target, the specification of the stipulated properties, and the employed verification principle. We give a representative overview of mechanisms to achieve modularity and compositionality along the three mentioned contexts and analyze how mechanisms in different contexts are related. In many verification frameworks one of the contexts carries the main burden. It is important to clarify these relations to understand the potential and limits of the different modularity mechanisms.

1 Introduction

Modularity and compositionality are core principles in all engineering fields and play a major role in verification approaches in Computer Science as well. While the two notions are sometimes used interchangeably, they relate to two slightly differing principles:

Compositionality is a way to break up a problem or system into subproblems or subsystems.

Modularity describes that a subsystem is a module: it has a clear interface and can be exchanged within the overall system with another module that has the same interface.

Henece, modularity is a desirable property of compositional *systems*, which is concerned with the design of interfaces between subsystems.

Modularity and compositionality in verification frameworks occur within different *contexts*. One can clearly distinguish three different contexts: the *model* that is the verification target, the *specification* of the stipulated properties, and the employed *verification principle*. We give a representative overview of mechanisms to achieve modularity and compositionality along the three mentioned contexts and analyze how mechanisms in different contexts are related. In many verification frameworks one of the contexts carries the main burden. It is important to clarify these relations to understand the potential and limits of the various modularity mechanisms.

System model: The language in which the verification target is formalized. This could be a mainstream programming language such as Java or C, a modeling language such as ABS [78] or ProMeLa [71], or a more abstract formalism such as the Actor model [68] or automata [6, 56, 75]. In any case, we assume that we have as a minimal requirement an executable language with a formal syntax and a precisely stated (though not necessarily formalized) runtime semantics.

System specification: The language in which system properties are expressed. In the simplest case, this means just assertions of Boolean expressions or even a finite set of fixed properties. But in most cases a specification is based on a more expressive logic, such as temporal or first-order logic. Even more expressive safety properties are possible with contract-based languages such as JML [91] or ACSL [15]. Finally, logical frameworks such as Coq [21] or Isabelle [102] permit not only to specify almost any property, but also the syntax and semantics of the system model.

Verification principle: Less obvious than the first two, this concers the verification methodology used to (dis-)prove properties of the system *model* expressed in the *specification* language. For example, state exploration together with abstraction refinement [62, 43] is a popular approach in model checking. Axiomatic approaches based on a calculus for a program logic are common in deductive verification [65]. Symbolic execution [29, 38, 86] is often the underlying principle in either [3, 24]. On the other hand, interactive theorem proving based on structural induction is the main verification principle employed in logical frameworks [21, 102] and inductive theorem proving of functional programs [36, 121].

In each context, mechanisms for modularity are expressed at differing levels of granularity. Before we discuss and analyze some representative modularity mechanisms, we can already state a few observations at a high level of abstraction:

1. The choice of modularity mechanism in different contexts is *not independent*. For example, a specification language with contracts does not make sense in combination with a system model that knows no procedures, symbolic execution cannot be used to prove properties of abstract programs.
2. Often one of the contexts is *dominant* over the others or can be considered as the starting point of the overall approach. In this case, choosing the modularity mechanism in the "lead" context determines the one in the others.
3. The *burden of modularity* may be unfairly allocated among different contexts: one can restrict an execution model to the extent that modular specification and verification become straightforward, such as the strong encapsulation of Din and Owe [54]. Or one can shift the burden to the power of the underlying verification approach, as the `CPAchecker` system [23] does. This may or may not be aligned with the dominant context.

2 Modularity and Composition Mechanisms: A Representative Collection

We follow the classification into the three contexts discussed above. Observe that artifacts belonging to different contexts are not necessarily separately formalized entities. For example, in interactive proof assistants the (abstract) syntax of the model, the specification language, and the verification approach are uniformly represented in higher-order formalisms, such type theory [21] and higher-order logic [102]; in dynamic logic [67], programs and their specifications both appear inside formulas, etc. Nevertheless, the context distinction is *conceptually* present and it is useful to make.

2.1 Model

Perhaps the central and most important modularity mechanism in programming is the *method*[3] abstraction. It is a pillar of modularity, because—in principle—one does not need to know the implementation details of a method in order to use it. Rather, it is sufficient to know under which *assumptions* it is supposed to work and what the intended *effects* (side effects and returned result) are.

Methods can be too fine-grained or too coarse-grained to modularize a program's behavior. This is especially true in concurrent systems. For example, a system model may provide full *data encapsulation* via objects [112] or *behavioral encapsulation* via atomic (non-interruptable) code segments [48]. Then all methods of an object *cooperate* to (re-)establish the *object invariant* at return from each call. The actor model is a point in case [4, 34, 54]. Vice versa, in languages such as C, where preemption and direct manipulation of the heap is permitted, one can neither abstract away from the implementation of a method, nor from its execution context.

Packages and modules in the sense of Java, C, etc., are important for compositionality at the level of the model, but they are limited to provide syntactic mechanisms for managing the namespace and help with disambuiguation. For verification purposes this is not enough, because it is essential to (de-)compose *behavior*.

Some abstract modeling languages have been designed with parallel composition operators that enjoy *algebraic* properties that make reasoning about correctness easy. Such *process calculi* include CCS [96], CSP [70], or the π-calculus [97, 98]. The downside is that their concurrency models are not realistic enough to permit efficient implementation. A different class of abstract models with "innate" modularity are *pure functional programs*. By construction, pure functions can be specified and verified independently of each other. Where modularity comes into play is when induction arguments in the proof of complex functions need to be decomposed to become automatically provable. This has led to the

[3] There are plethora of synonyms for the same concept in different contexts: function, procedure, routine, etc. In this paper we use the term *method* without committing to a specific execution model.

development of such techniques as *rippling* [37] or *generalization* [119] that help with lemma discovery [77].

Some of the abstract concurrency models use the encapsulation inherent to distributed systems to provide modularity. Choreographies [7, 31, 105] implement a global view for message passing between services, e.g., in business protocols [39]. A choreography is used as the *endpoint projection* to generate code for single services. This code is guaranteed by construction to realize the order of messages in the choreography. Similarly, orchestration [105] describes a central entity that controls messages between services. It is crucial that the entities are encapsulated and have no other way of communicating. Both notions are deeply connected and choreographies can be used to derive an orchestrator [94]. Choreographies may either commit to one interaction style or mix different interaction styles (e.g., synchronous and asychronous communication) [10]. The aforementioned actor model is another concurrency model that uses encapsulated distribution to provide modularity.

In software product line engineering (SPLE) modularity is expressed along the composition of features as requested by a product specification [107]. Specifically, in *delta-oriented programming* [109] the implementation of features is associated with *code deltas* specifying the implementation of a feature relative to a given core. A general overview of feature-oriented implementation techniques is [8].

In system engineering and hybrid systems *continuous* state changes, in addition to calls and discrete state changes, are modeled. One compositional technique to do so are components [61]. Components strictly distinguish between the internal *behavior model* and the *interface model* that connects different components via their ports. The interface model can be synchronous or asynchronous and different instances of behavior models are supported. In the area of hybrid systems, (hybrid) I/O automata [93] also offer a basic composition mechanism as a low-level device, albeit only via modeling synchronization on transitions.

2.2 Specification

In the simplest case, a specification consists merely of a generic property, for example, "absence of deadlocks", "(normal) termination", etc. Model-specific properties are mostly expressed in suitable logics. In a basic setup, logical *assertions* are placed at certain locations in the model, where they must hold in any run. When an assertion appears at the syntactic end of a method it functions as a postcondition. Dual to assertions, one can instrument a model with *assumptions*. These are properties that can be assumed to hold in the execution state where they occur. An assumption placed at the syntactic beginning of a program works as a precondition. Hoare logic [69] is based on assumption-program-assertion triples.

Program logics specify not merely a single computation state, but express properties of whole runs and thus can relate multiple execution states. They include temporal logic [12] and dynamic logic [67]. Generic properties, assertions, assumptions, and program logics do not provide any support for modularity in

themselves, but assumptions and assertions can be used as elements of modular specification formalisms.

A simple form of modularity are *invariants*. They can take the form of object or loop invariants, but in either case the idea is the same: *assume* that a certain property (the "invariant" I) holds initially. If—under this assumption—the execution of a given model M *asserts* the invariant upon termination in each run, then I is an invariant for M. A simple induction yields that I is also an invariant for an arbitrary number of subsequent executions of M. Now imagine that M is a loop body and I is asserted at the start of the loop. Or M is any method of an object and all constructors assert I. Then I holds whenever the loop or a call to one of the object's methods terminates, respectively. This allows to replace the behavior of a loop or of an object by its invariant during verification and it constitutes a base line of modularity.

Most contemporary deductive verification frameworks (for example, [3, 26, 76, 87, 118]) use a specification language based on the notion of a *method contract*. First introduced by Meyer [95] as *design-by-contract* in the context of runtime verification, a method contract comprises the assumptions under which a method is supposed to work correctly, together with an assertion of its intended effects (side effects and returned result). Thus, contracts can be composed from the building blocks "assertion" and "assumption" over a logical language (usually, typed first-order logic). The requirements that need to hold for a method contract to enter into force are its *precondition*. The stipulated final computation state (including the returned result) after a method terminates is its *postcondition*. The memory locations a method can read are called its *footprint*, the memory locations it can write to are called its *frame*.

Method contracts are a central device to achieve a degree of modularity in specification, because they can characterize the effect of a method call without actually having to analyze the called method. This is essential to make verification of large programs feasible: clearly a program with hundreds of method declarations cannot be analyzed by inlining method calls. Enforcing modularity here also enables local re-verification: changes in one method require one to re-verify the changed method, not the whole system. In the case of recursively defined methods, contracts even enable verification in the first place. In some cases, in particular, in dynamic analyses, the notion of contract is only *implicit*, for example [64] speak of *structural properties* and [60] of *summaries*.

While it is obvious that method contracts must describe the behavior of the called method (the *caller*'s perspective), it is less obvious that one must pay as well attention to the call context (the *callee*'s perspective). The problem arises, whenever the frame or footprint of a method include the heap. Since the callee cannot know in which heap state it is called, the pre- and postcondition have to be expressed so that they are valid in arbitrary states. This means, for example, that the effects of a method on an *unknown* heap that may intersect with its frame and footprint have to be described correctly. A number of techniques to achieve this have been developed, including dynamic frames [85, 111], ownership types [42, 51], boxing [92, 106], and separation logic [103].

The situation worsens in the case of concurrent programs, because of task interleaving. This led to mechansims such as fractional permissions [30], concurrent separation logic [32], and combinations thereof [28], for low-level concurrency, and to context-aware contracts [83] for concurrency with atomic segments. Assumption-guarantee reasoning is not necessarily bound to method contracts, pairs of pre- and postconditions can also be used to specify processes [99].

It has to be stressed that while various framing theories make it possible to achieve a certain degree of modularity when specifying complex, heap-manipulating software [5, 49] this does not mean that the approach is practical yet: often specifications become considerably longer than the specified model and are harder to understand [16].

From a feature-oriented SPLE perspective it makes sense to *compose* contracts. Specification deltas [66] reuse contract elements in analogy to code deltas, but this works smoothly only when behavioral subtyping is assumed. That this is generally not the case is shown in [116], which also contains an overview of feature-oriented contract composition techniques.

Contracts specify the behavior of a method at its endpoints. In particular to specify concurrent models, it may be necessary to expose some of the intermediate behavior. Therefore, it is a natural idea to generalize contracts to symbolic traces [53, 81, 115]. It remains to be seen, however, whether this leads to improved modularity.

Component contracts were studied to specify the interface level of components [20]. These contracts are also based on assumption-guarantee reasoning, and specify what a component must guarantee *to* the environment and what it can rely on *from* the environment. Component contracts differ from method contracts, as they specify the continuously changing ports of a component at interface level. Component contracts abstract not only from the concrete behavior of a component, but also from the language of its implementation. This allows contracts to inherit the compositional properties of components through contract operators [19], but limits the specifications to boolean assertions over ports. Interface automata [47] are a formalism similar to the aforementioned I/O automata that specify the temporal behavior of automata and have a composition mechanism compatible with open systems: They specify the expected behavior of the environment of an I/O automaton.

One version of program development by step-wise *refinement* works by specifying a series of ever more precise abstract machines [1, 2, 110, 27] that are finally translated into executable code. Such a development can be seen as a series of *modular* specifications.

2.3 Verification

Modularity can occur in several places in the verification context, either by decomposition following another context, e.g., following the structure of contracts or methods, or decompose a problem that is neither specified modularly nor executed modularly.

With *axiomatic decomposition* we denote a verification approach that allows to decompose a verification task into subtasks by way of a decomposition axiom. The *frame rule* of separation logic [103] that allows to localize heap reasoning is a well-known example. Another example can be found in early work on modular verification of simple concurrent programs (without heaps and method calls) in the form of *assumption-guarantee* reasoning [80] and its predecessor, the *Owicki-Gries* composition axiom [104][4]. Similarly, composition of proofs based on the communication between processes has been axiomatized for synchronization based formalisms, such as ADA [9, 59]. These principles are implicitly present in many contemporary approaches as well and have been frequently generalized (for example, [57, 90]). Being based on axiomatic decomposition, they lend themselves well to deductive verification. In fact, contract-based specification can be seen as a specification language well aligned with assumption-guarantee reasoning.

Not in each case is the underlying logical framework expressive enough to justify a decomposition step. For example, [54, 84] prove a meta theorem justifying the problem decomposition. This is more a matter of taste or perhaps the desired degree of mechanization, because it is often possible to justify decomposition in a suitably expressive logic [81, 101]. The limitation of axiomatic decomposition is often that the decomposition theorem holds only under certain constraints, which becomes an issue with respect to scalability to complex target languages.

A different decomposition technique is *projection*. Session Types [72, 73] are a behavioral type discipline [7, 74] using global types to describe protocols. Global types are projected onto a role—this generates a local type for each protocol endpoint. It is similar to projection of choreographies on the modeling level [40] (cf. Sect. 2.1): indeed, global types are used as specifications of choreographies [39]. The target language, where type checking happens, are the local types. Projection is designed in a way that enforces further encapsulation in the concurrency model to ensure modularity: the main verification step is a fully automatic argument that composes adherence to the local types to adherence of the whole system to the global types. Not every global type can be projected and projection depends very much on the concurrency model of the target language. In particular, the notion of an endpoint may correspond to a fixed entity in the concurrency model (for example, for actors [82]), but does not need not to do so (for example, for the π-calculus [73]).

Correctness-by-construction [52, 63, 11, 89, 113] is the step-wise development of (simple, usually imperative) programs in a series of refinement steps. Each step is justified in a suitable program logic, so that this can be seen as a modular verification strategy.

Abstraction is a general principle to approximate the behavior of a model and its datastructures during execution. This approach was pioneered by Cousot & Cousot [45] as *abstract interpretation* of programs. It allows to abstract away

[4] It is worth noting that the original Owicki-Gries composition axiom verifies all involved processes without encapsulation, i.e., changing one process requires to reprove the composition as well.

Table 1. Mechanisms to achieve modularity and compositionality in different contexts

Context	Model	Specification	Verification
Baseline	unstructured code	assume, assert	**intermediate assertion**, **cut**, **interpolant**, **abstraction**
Mechanism	method, loop	**contract**, **framing**, **loop invariant**	axiomatic decomposition, abstract execution
	object, actor, atomic segment	**invariant**, **symbolic trace**	meta composition
	feature	contract composition	**proof composition**
	(generic)	session type	**projection**
	choreography	(generic)	projection
	component	component contract	(generic)
	process calculus	(generic)	axiomatic decomposition
	functional program imperative program	**abstract machine** **assume**, **assert**	refinement
	pure functions	assume, assert	rippling, generalization
	(verification-specific)	(verification-specific)	proof reuse

from intricate data structures or complex behavior, so that a specified property is easier to prove. Of course, it can happen that the property does not hold anymore for the abstract version. In this case, it is desirable to find the exact degree of abstraction where it still holds. *Abstraction refinement* [43, 62] allows to determine it in an automated manner. Abstraction is not a modularization technique *per se*, but a base line verification principle.

This is in contrast to *abstract execution* [114], where a program with abstract statements is symbolically executed, so that whatever can be proven about the abstract program holds as well for any of its (legal) instances. It allows to decouple programs from their execution contexts, because the latter can be specified by abstract symbols.

It is also possible to modularize verification problems at the level of proofs. A well-known example is TLA+ [41], where proofs are arranged hierarchically.[5] Similarly, proofs for hybrid systems can be composed if the underlying structure has a component-like structure [100]. *Proof reuse* can also be seen as modularization, for example, lazy symbolic execution [35, 55], proof repositories [33], or proof adaptation [18, 108, 120]. In the context of *family-based* verification approaches in SPLE, a number of proof composition techniques have been explored [14, 50, 117]. Several of these techniques do not work directly on proofs, which tends to be brittle, but on more abstract representations such as contracts or proof scripts. Even so, these techniques are necessarily tied to a specific verification approach.

[5] It is also possible to view refinement-based approaches from this angle.

3 Alignment of Context and the Burden of Modularity

In Table 1 we summarize some of the modularity mechanisms discussed above. In addition, we attempt to relate them across different contexts. Obviously, this correspondence is neither precise nor exhaustive, but should be seen as a basis for more in-depth investigations or for discussion. In each row we highlight the context that carries the burden of modularity in boldface. Under carrying the burden we mean the burden to provide the modularity that is used by the other contexts.

One can instantiate the table to a wide range of established verification approaches. Just as an example, deductive verification [65] is typically built around the notions of contract and framing of structured pieces of code (blocks, methods, loop bodies).

4 Track Contributions

We briefly discuss the contributions of the ISoLA 2020 track on "Modularity and (De-)composition in Verification" in the light of the classification above and mention where in an approach the burden of modularity lies.

4.1 Modularity in the Context of the Model

Coto et al. [44] (*On Component Testing Message-Passing Applications*) address the problem of generating tests for the components of systems in which the components (or participants) communicate via asynchronous message passing (but where the message buffers do not preserve the order of the messages). The correct coordination of the components is specified by means of global choreographies. Following the (top-down) correctness-by-construction principle, the component test suites are obtained by projecting the choreography suitably along the (interfaces of the) components. The generated tests are guaranteed to be suitable, in the sense that every valid implementation of the given component necessarily passes them. The authors discuss a number of aspects of the considered problem, such as test oracles, efficiency of test generation, and coverage criteria.

4.2 Modularity in the Context of the Specification

Barbanera et al. [13] (*Composing Communicating Systems, Synchronously*) investigate the preservation of generic behavioural properties, and in particular deadlock freedom, under the synchronous composition of systems of communicating finite state machines. A composability condition, two structural constraints, and two types of composition are defined, for which it is proved that composing composable systems satisfying the structural constraints preserves deadlock freedom. The authors argue that the same reasoning can be applied to other generic behavioural properties such as lock freedom and liveness.

Beckert et al. [17] (*Modular Verification of JML Contracts Using Bounded Model Checking*) aim to connect the worlds of contract-based deductive verification (DV) with the one of bounded software model checking (BMC). The burden is in translating JML-annotated JAVA into plain JAVA with asserts and assumes. The latter then can serve as input to the bounded model checker JBMC. Obviously, the translation must be parameterized with a (loop and recursion) bound. Technically, the problem of replacing quantifiers by non-deterministic assignments is a central issue. The translation creates the opportunity to run JBMC on JML-annotated programs. In addition to better efficiency and higher automation, this opens up interesting new scenarios for collaboration of DV and MC the authors point out. It is worth to point out that the suggested tool combination perfectly aligns with the case for integration of verification approaches brought forward in [65].

Further, in the domain of Software Product Line Engineering (SPLE), Damiani et al. [46] (*On Slicing Software Product Line Signatures*) present an abstraction and *decomposition* technique based on slicing. A slice of SPL Signature (SPLS) for some feature set $\mathcal{F}$ is a product line that contains neither implementation details of its classes nor products that depend on features outside $\mathcal{F}$. The paper defines such slices and discusses the challenges for an algorithm that computes the slice manually and efficiently. As the main driver of this approach is the specified feature set $\mathcal{F}$, the burden of modularity lies with the specification and the check that the slice is given correctly.

Johnsen et al. [79] (*Assumption-Commitment Types for Resource Management in Virtually Timed Ambients*) introduce a type system for resource management in the context of nested virtualization. The type system is based on effect/coeffect pairs that specify how much resources a process may consume from its context and how much it must offer to its child process. This allows to type check a process in isolation by specifying a resource interface. Nonetheless, the burden of modularity is only partially with the specification: as the effect/-coeffect pairs are derived for the inner processes automatically, it lies also with the verification. However, this is enabled by the structure of the specification.

4.3 Modularity in the Context of Verification

Filliâtre and Paskevich [58] (*Abstraction and Genericity in Why3*) argue that any approach invented for modularity in programming can also be adapted to program verification. The purpose of their contribution is to show how they achieve this in WhyML, the language of the program verification tool Why3. WhyML uses a single concept of module, a collection of abstract and concrete declarations, and a basic operation of cloning which instantiates a module with respect to a given partial substitution, while verifying its soundness. This mechanism brings into WhyML both abstraction and genericity, which the authors illustrate on a small verified Bloom filter implementation.

Then, Beyer and Wehrheim [25] (*Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework*) give a classification of combinations of verification approaches. The focus is on black-box integration

through the exchange of artifacts between multiple tools. The paper discusses the exchange format and different roles that tools can play in a cooperative verification framework. The approach described here places the burden of modularity firmly on the verification.

Beyer and Kanav [22] (*An Interface Theory for Program Verification*) take a verification-centric view: the set of all behaviors of a program P is viewed as a behavioral interface I_P. Verification is then recast as a theory of approximation of behavioral interfaces. As pointed out above, different viewpoints on verification are relevant, because they suggest different decomposition strategies of the verification process. Ideally, this leads to the identification and optimization of core reasoning tasks or to new cooperation strategies. The present paper suggests to use over- and underapproximations of the target program as a uniform mechanism to exchange information between different tools in the form of behavioral interfaces. This leads naturally to a decomposition strategy inspired by gradual specification refinement. A number of known verification approaches are characterized from this angle. The paper provides a uniform view of existing work that invites to think about new ways by which verification tools may cooperate.

Finally, Knüppel et al. [88] (*Scaling Correctness-by-Construction*, CbC) suggest an architectural framework for contract-based, CbC-style program development. While both the target language as well as the current implementation are at the proof-of-concept level, it is a promising start. The CbC approach was until recently characterized by an almost complete lack of tools. The paper describes steps towards scaling mechanized CbC development to more complex programs and to establish a repository of reusable off-the-shelf components. To this end, the authors present a formal framework and open-source tool named ArchiCorC. There, a developer models software components in UML-style with required and provided interfaces. Interface methods are associated to specification contracts and mapped to verified CbC implementations. A code generator backend then emits executable Java source code.

5 Conclusion

In this paper we gave an overview of modularity in verification and attempted a classification. Hopefully, this can serve as a starting point for the structural use of modularity principles in verification: First, we hope that the classification helps the research community to transfer ideas between subfields by guiding discussions and reengineering approaches. Abstraction from the technical details of modularity allows the underlying *ideas* to be carried over to fields where the original system is not directly applicable. Second, we believe that the classification has the potential to motivate systematic search for *new* modularity mechanisms. Lastly, a classification may shed light on the structure of existing systems and so may guide their eventual reengineering for an increase of modularity.

Acknowledgements We thank Marieke Huisman and Wolfgang Ahrendt for their very constructive and valuable feedback on a draft.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.
2. J.-R. Abrial. *Modeling in Event-B — System and Software Engineering*. Cambridge University Press, 2010.
3. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *LNCS*. Springer, 2016.
4. W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. K. Breitman and A. Cavalcanti, editors, *Formal Methods and Software Engineering, 11th Intl. Conf. on Formal Engineering Methods, ICFEM, Rio de Janeiro, Brazil*, volume 5885 of *LNCS*, pages 387–406. Springer, 2009.
5. E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third Intl. Conference, VSTTE, Edinburgh, UK*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010.
6. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Comp. Sci.*, 126(2):183–235, 1994.
7. D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.
8. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
9. K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3):359–385, 1980.
10. F. Arbab, L. Cruz-Filipe, S. Jongmans, and F. Montesi. Connectors meet choreographies. *CoRR*, abs/1804.08976, 2018.
11. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.
12. B. Banieqbal, H. Barringer, and A. Pnueli, editors. *Temporal Logic in Specification*. Springer, 1987.
13. F. Barbanera, I. Lanese, and E. Tuosto. Composing communicating systems, synchronously. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
14. D. S. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. of Universal Computer Science*, 14(12):2059–2082, 2008.
15. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA LIST and INRIA, 1.4 edition, 2010.
16. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons learned from microkernel verification – specification is the new bottleneck. In F. Cassez, R. Huuck, G. Klein, and B. Schlich, editors, *Proc. 7th Conference on Systems Software Verification*, volume 102 of *EPTCS*, pages 18–32, 2012.
17. B. Beckert, M. Kirsten, J. Klamroth, and M. Ulbrich. Modular verification of JML contracts using bounded model checking. In T. Margaria and B. Steffen,

editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
18. B. Beckert and V. Klebanov. Proof reuse for deductive program verification. In *2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM), Beijing, China*, pages 77–86. IEEE Computer Society, 2004.
19. A. Benveniste, B. Caillaud, H. Elmqvist, K. Ghorbal, M. Otter, and M. Pouzet. Multi-mode DAE models - challenges, theory and implementation. In B. Steffen and G. J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 283–310. Springer, 2019.
20. A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 6th Intl. Symp., FMCO, Amsterdam, The Netherlands, Revised Lectures*, volume 5382 of *LNCS*, pages 200–225. Springer, 2007.
21. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
22. D. Beyer and S. Kanav. An interface theory for program verification (position paper). In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
23. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification, 23rd Intl. Conf., CAV, Snowbird, UT, USA*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.
24. D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 7th International Symposium (ISoLA), Part I, Corfu, Greece*, volume 9952 of *LNCS*, pages 195–211. Springer, Oct. 2016.
25. D. Beyer and H. Wehrheim. Verification artifacts in cooperative verification: Survey and unifying component framework. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
26. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In *IFM*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.
27. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
28. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 259–270. ACM, 2005.
29. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.
30. J. Boyland. Fractional permissions. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 270–288. Springer, 2013.

31. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In M. Lumpe and W. Vanderperren, editors, *Software Composition, 6th Int.l Symp., SC, Braga, Portugal, Revised Selected Papers*, volume 4829 of *LNCS*, pages 34–50. Springer, 2007.
32. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.
33. R. Bubel, F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, and I. C. Yu. Proof repositories for compositional verification of evolving software systems. In *Foundations for Mastering Change (FoMaC) I*, volume 9960 of *LNCS*, pages 130–156. Springer, 2016.
34. R. Bubel, C. C. Din, R. Hähnle, and K. Nakata. A dynamic logic with traces and coinduction. In H. D. Nivelle, editor, *Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Wroclaw, Poland*, volume 9323 of *LNCS*, pages 303–318. Springer, 2015.
35. R. Bubel, R. Hähnle, and M. Pelevina. Fully abstract operation contracts. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece*, volume 8803 of *LNCS*, pages 120–134. Springer, Oct. 2014.
36. A. Bundy. The automation of proof by mathematical induction. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 845–911. Elsevier and MIT Press, 2001.
37. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, June 2005.
38. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
39. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012.
40. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty sessions. In R. Bruni and J. Dingel, editors, *Formal Techniques for Distributed Systems: Joint 13th IFIP WG 6.1 Intl. Conf., FMOODS, and 31st IFIP WG 6.1 Intl. Conf., FORTE, Reykjavik, Iceland*, volume 6722 of *LNCS*, pages 1–28. Springer, 2011.
41. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA+ proof system. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *Proc. LPAR Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
42. D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 15–58. Springer, 2013.
43. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, Chicago/IL, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
44. A. Coto, R. Guanciale, and E. Tuosto. On component testing message-passing applications. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.

45. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, Jan. 1977.
46. F. Damiani, M. Lienhardt, and L. Paolini. On slicing software product line signatures. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
47. L. de Alfaro and T. A. Henzinger. Interface automata. In A. M. Tjoa and V. Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 109–120. ACM, 2001.
48. F. de Boer, C. C. Din, K. Fernandez-Reyes, R. Hähnle, L. Henrio, E. B. Johnsen, E. Khamespanah, J. Rochas, V. Serbanescu, M. Sirjani, and A. M. Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5):76:1–76:39, Oct. 2017. Article 76.
49. S. De Gouw, F. S. De Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. Verifying OpenJDK's sort method for generic collections. *J. Automated Reasoning*, 62(6), 2019.
50. B. Delaware, W. R. Cook, and D. S. Batory. Product lines of theorems. In C. V. Lopes and K. Fisher, editors, *Proc. 26th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Portland, OR, USA*, pages 595–608. ACM, 2011.
51. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
52. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
53. C. C. Din, R. Hähnle, E. B. Johnsen, K. I. Pun, and S. L. Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In *TABLEAUX*, volume 10501 of *LNCS*, pages 22–43. Springer, 2017.
54. C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.
55. J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.
56. E. A. Emerson. Automata, tableaux and temporal logics (extended abstract). In *Proceedings Conference on Logics of Programs, Brooklyn*, LNCS 193, pages 79–87. Springer, 1985.
57. X. Feng. Local rely-guarantee reasoning. In Z. Shao and B. C. Pierce, editors, *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Savannah, GA, USA*, pages 315–327. ACM, 2009.
58. J.-C. Filliâtre and A. Paskevich. Abstraction and genericity in Why3. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
59. R. Gerth and W. P. de Roever. A proof system for concurrent ADA programs. *Sci. Comput. Program.*, 4(2):159–204, 1984.
60. P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In M. B. Dwyer and F. Tip, editors, *Proc. 20th Intl. Symp. on Software Testing and Analysis, ISSTA, Toronto, Canada*, pages 23–33. ACM, 2011.

61. G. Gößler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1–3):161–183, 2005.
62. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.
63. D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer, 1981.
64. D. Gurov and M. Huisman. Reducing behavioural to structural properties of programs with procedures. *Theoretical Computer Science*, 480:69–103, 2013.
65. R. Hähnle and M. Huisman. Deductive verification: from pen-and-paper proofs to industrial tools. In B. Steffen and G. Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, volume 10000 of *LNCS*, pages 345–373. Springer, 2019.
66. R. Hähnle and I. Schaefer. A Liskov principle for delta-oriented programming. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change — 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *LNCS*, pages 32–46. Springer, Oct. 2012.
67. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, Oct. 2000.
68. C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.
69. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 583, Oct. 1969.
70. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
71. G. J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.
72. K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR, 4th Intl. Conf. on Concurrency Theory, Hildesheim, Germany*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
73. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. of the ACM*, 63(1):9:1–9:67, 2016.
74. H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49(1):3:1–3:36, Apr. 2016.
75. M. Isberner, F. Howar, and B. Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1–2):65–98, 2014.
76. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.
77. M. Johansson, L. Dixon, and A. Bundy. Dynamic rippling, middle-out reasoning and lemma discovery. In S. Siegler and N. Wasser, editors, *Verification, Induction, Termination Analysis: Festschrift for Christoph Walther on the Occasion of His 60th Birthday*, volume 6463 of *LNCS*, pages 102–116. Springer, 2010.
78. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. de Boer, and

M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer-Verlag, 2011.
79. E. B. Johnsen, M. Steffen, and J. B. Stumpf. Assumption-commitment types for resource management in virtually timed ambients. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
80. C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 321–332. North-Holland/IFIP, 1983.
81. E. Kamburjan. Behavioral program logic. In *TABLEAUX*, volume 11714 of *LNCS*, pages 391–408. Springer, 2019.
82. E. Kamburjan and T. Chen. Stateful behavioral types for active objects. In C. A. Furia and K. Winter, editors, *Integrated Formal Methods: 14th Intl. Conf., IFM, Maynooth, Ireland*, volume 11023 of *LNCS*, pages 214–235. Springer, 2018.
83. E. Kamburjan, C. C. Din, R. Hähnle, and E. B. Johnsen. Behavioral contracts for cooperative scheduling. In W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, and M. Ulbrich, editors, *Deductive Verification: The State of the Future*, volume 12345 of *LNCS*. Springer, 2020.
84. E. Kamburjan, R. Hähnle, and S. Schön. Formal modeling and analysis of railway operations with Active Objects. *Science of Computer Programming*, 166:167–193, Nov. 2018.
85. I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, 2011.
86. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
87. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: a software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
88. A. Knüppel, T. Runge, and I. Schaefer. Scaling correctness-by-construction. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, Oct. 2020. In this proceedings.
89. D. G. Kourie and B. W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012.
90. O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming: 42nd Intl. Coll., ICALP, Kyoto, Japan*, volume 9135 of *LNCS*, pages 311–323. Springer, 2015.
91. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, May 2013. Draft revision 2344.
92. K. R. M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In N. Shankar and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference (VSTTE), Toronto, Canada*, volume 5295 of *LNCS*, pages 192–208. Springer, 2008.
93. N. A. Lynch, R. Segala, F. W. Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems*

III: Verification and Control, DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, Rutgers University, New Brunswick, NJ, USA, volume 1066 of *LNCS*, pages 496–510. Springer, 1995.
94. S. McIlvenna, M. Dumas, and M. T. Wynn. Synthesis of orchestrators from service choreographies. In M. Kirchberg and S. Link, editors, *Conceptual Modelling 2009, Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM), Wellington, New Zealand*, volume 96 of *CRPIT*, pages 129–138. Australian Computer Society, 2009.
95. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
96. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
97. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.
98. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.
99. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
100. A. Müller, S. Mitsch, W. Retschitzegger, W. Schwinger, and A. Platzer. Tactical contract composition for hybrid system component verification. *Int. J. Softw. Tools Technol. Transf.*, 20(6):615–643, 2018.
101. L. P. Nieto. The Rely-Guarantee Method in Isabelle/HOL. In P. Degano, editor, *Programming Languages and Systems, 12th European Symp. on Programming, ESOP, Warsaw, Poland*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.
102. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
103. P. W. O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019.
104. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
105. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
106. A. Poetzsch-Heffter and J. Schäfer. Modular specification of encapsulated object-oriented components. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proc. 4th Intl. Symp. Methods for Components and Objects, (FMCO), Amsterdam, The Netherlands, Revised Lectures*, volume 4111 of *LNCS*, pages 313–341. Springer-Verlag, 2006.
107. K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
108. W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 13th Conf., Bombay, India*, volume 761 of *LNCS*, pages 284–293. Springer, 1993.
109. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond: 14th Intl. Conf., SPLC, Jeju Island, South Korea*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.
110. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *Journal of Universal Computer Science*, 3(4):377–412, 1997.
111. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in java dynamic logic. In B. Beckert and C. Marché, editors, *Formal Verification of Object-Oriented*

Software, Intl. Conf. FoVeOOS, Paris, France, Revised Selected Papers, volume 6528 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.
112. M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informatica*, 63(4):385–410, 2004.
113. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
114. D. Steinhöfel and R. Hähnle. Abstract execution. In A. McIver and M. ter Beek, editors, *Formal Methods: The Next 30 Years*, volume 11800 of *LNCS*, pages 319–336. Springer, 2019.
115. D. Steinhöfel and R. Hähnle. The trace modality. In A. Baltag and L. S. Barbosa, editors, *2nd Intl. Workshop on Dynamic Logic: New Trends and Applications*, volume 12005 of *LNCS*, pages 124–140, Cham, Jan. 2020. Springer.
116. T. Thüm, A. Knüppel, S. Krüger, S. Bolle, and I. Schaefer. Feature-oriented contract composition. *Journal of Systems and Software*, 152:83–107, 2019.
117. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Fourth IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST, Berlin, Germany, Workshop Proc.*, pages 270–277. IEEE Computer Society, 2011.
118. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS, London, UK*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015.
119. P. Urso and E. Kounalis. Sound generalizations in mathematical induction. *Theoretical Computer Science*, 323(1-3):443–471, 2004.
120. C. Walther and T. Kolbe. Proving theorems by reuse. *Artificial Intelligence*, 116(1–2):17–66, 2000.
121. C. Walther and S. Schweitzer. About VeriFun. In F. Baader, editor, *Automated Deduction, 19th International Conference on Automated Deduction, Miami Beach, USA*, volume 2741 of *LNCS*, pages 322–327. Springer, 2003.