Report from Dagstuhl Seminar 12511

Divide and Conquer: the Quest for Compositional Design and Analysis

Edited by

Marieke Huisman[1], Barbara Jobstmann[2], Ina Schaefer[3], and Marielle Stoelinga[4]

1 **University of Twente, NL,** Marieke.Huisman@ewi.utwente.nl
2 **VERIMAG – Gières, FR,** Barbara.Jobstmann@imag.fr
3 **TU Braunschweig, DE,** i.schaefer@tu-braunschweig.de
4 **University of Twente, NL,** Marielle.Stoelinga@ewi.utwente.nl

Abstract

On December 16 to 21, the Dagstuhl seminar *Divide and Conquer: the Quest for Compositional Design and Analysis* was organized. Topic was the *compositionality*, a central theme in computer science, but its applications, methods, techniques are scattered around many different disciplines. Therefore, this workshop brought together scientists from different disciplines, including deductive verification, model checking, software product lines, component interfaces.

Seminar 16.–21. December, 2012 – www.dagstuhl.de/12511
1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3 Logics and Meanings of Programs, F.3.1 Specifying and Verifying and Reasoning about Programs
Keywords and phrases Algorithms, Design, Languages, Theory, Verification
Digital Object Identifier 10.4230/DagRep.2.12.64

1 Executive Summary

Marieke Huisman
Barbara Jobstmann
Ina Schaefer
Marielle Stoelinga

Compositionality is a key concept in computer science: only by breaking down a large system into smaller pieces, we can build today's complex software and hardware systems. The same holds true for verification and analysis: realistic systems can only be analyzed by chopping them up into smaller parts. Thus, compositionality has been widely studied in various different settings, and by different communities: people in programming languages, software verification, and model checking have all come up with their own techniques and solutions.

Thus, the goal of this workshop has been to bring together these fields and communities, so that they can learn from and cross-fertilize each other. We have succeeded in doing so: through three extensive tutorials, longer and shorter presentations, and working sessions, researchers from different areas have learned about each others problems, techniques, and approaches.

The scientific programme was built around four corners stones

1. Personal introductions.
2. Three well-received tutorials:
   - Compositional programming by Oscar Nierstrasz
   - Compositional verification by Arnd Poetzsch-Heffter
   - Compositional modelling by Arend Rensink
3. Regular presentations, presenting in-depth technical knowlegde on:
   - Verification of programming languages
   - Automatic synthesis
   - Interface theories
   - Model checking
   - Contract-based design
   - Software product lines
4. Working group sessions:
   - Working group on software product lines
   - Working group on Benchmark for Industrial Verification/Synthesis Problems
   - Working group on Modular Full Functional Specification and Verification of C and Java programs that Perform I/O
   - Model checking vd deducutive verification
   - Compositional Synthesis of Reactive Systems

2 Table of Contents

3 Detailed programme

3.1 Personal introductions

We started out by a three rounds of personal introductions of the participants. Each participant was asked to introduced him or herself in two minutes, focussing on scientific interests. Also, we asked participants to classify themselves according to three basic scientific disciplines, being modeling, verification and/or programming languages. In our opinion, these introductions really helped to get to know each other, to easy communication and discussion, and "break the ice."

3.2 Tutorials

In order to set a common ground between the various fields, we asked three renown experts to give a tutorial, explaining the basic concepts, methods, and techniques in their field. In our opinion, these tutorials were a success, since their quality was excellent, and they were very well received by the participants.

3.2.1 Tutorial on compositional programming by Oscar Nierstrasz

Oscar Nierstrasz started out by a historical overview of compositional techniques in programming languages — that we all know, but were not aware of their historic origins. Then moved to current techniques in compositional programming, and challenges for the future.

3.2.2 Tutorial on compositional verification by Arnd Poetzsch-Heffter

Poetzsch-Heffter started with the presentation of basic principles of compositional verification and suggested to categorize techniques for compositional verification according to four dimensions: (A) What are the components? How are they represented? (B) What are the composition mechanisms? (C) What properties are addressed? How are they specified? (D) What kinds of verification techniques are used? In the main part, the tutorial took a closer look at four different settings to discuss important aspects in the huge space of compositional verification: (1.) Maximal models for model checking step-synchronous Moore machines (2.) Model checking control-flow properties of sequential procedural programs (3.) Modular state-based verification of object-oriented programs (4.) Assume-guarantee reasoning for communicating processes

3.2.3 Tutorial on compositional modelling by Arend Rensink

This presentation formulated a simple formal framework in which some of the essential aspects of compositionality come together. The most important lesson is that the composition operator should be compatible with a notion of abstraction, or semantics, encapsulating the conceptual understanding of the objects under construction. If this compatibility fails, the common solution is to augment the abstraction by putting more information into it. This kind of augmentation is, in fact, the same kind of step as strengthening the induction hypothesis in an inductive proof.

The framework is tested, with varying degrees of success, against a number of cases of composition from different domains, ranging from bisimilarity minimisation to testing and subclassing (seen as a composition operator). Some of these cases nicely illustrate the principle of augmentation; others provide examples where compositionality simply fails to hold.

4 Overview of Talks

4.1 Compositional verification and semantics of concurrent/distributed objects

Wolfgang Ahrendt (Chalmers UT – Göteborg, SE)

We present a semantics, calculus, and system for compositional verification of an object-oriented modelling language for concurrent distributed applications. The system is an instance of KeY, a framework for object-oriented software verification, which has so far been applied foremost to sequential Java. The presented system addresses functional correctness of models featuring local cooperative thread parallelism and global communication via asynchronous method calls. The calculus heavily operates on communication histories specified by interfaces. We also present adenotational semantics and an assumption-commitment style semantics of the logic.

4.2 A Coinductive Big-step Semantics for Distributed Concurrent Objects

Wolfgang Ahrendt (Chalmers UT – Göteborg, SE)

We present a fully compositional big-step operational semantics for globally distributed and locally concurrent objects. The semantics captures both, terminating and non-terminating behaviour. We construct thread histories independently, non-deterministically guessing effects of other threads/objects. Then thread histories are merged to object history, which then are merged to system histories.

4.3 Glue synthesis in BIP

Simon Bliudze (EPFL – Lausanne, CH)

BIP (Behaviour, Interaction, Priority) is a framework for the component-based design and analysis of real-time embedded systems. It has been successfully used for modelling and analysis of a variety of case studies and applications, such as performance evaluation, modelling and analysis of Tiny OS-based wireless sensor network applications, construction and verification of a robotic system. The main characteristic feature of BIP is the clear separation of component behaviour (sequential computation) and coordination. The latter is realised through memory less "glue" consisting of an interaction model defining synchronisations among components and a priority model used for conflict resolution and defining scheduling policies. In this talk, I will briefly discuss automatic synthesis of glue from boolean constraints representing a certain type of safety properties made possible due to the above-mentioned separation of concerns principle.

4.4 Defining a general abstract notion of component

Simon Bliudze (EPFL – Lausanne, CH)

Component-based design generally relies on a clear separation between the mechanisms used to design atomic components and those used to define the "glue" assembling these into higher-level compound components. Different component-based design approaches have different models for components and different – often ad-hoc – glue operators. Defining the appropriate glue for agiven component model and studying glue properties such as composition and interference, requires a formal and generic definition of the notion of glue. The necessity of such definition becomes even clearer when one tries to compare different component frameworks. However, defining a formal generic notion of glue requires first a formal generic notion of component. In a recent paper (http://rvg.web.cse.unsw.edu.au/eptcs/paper.cgi?ICE2012.6), we have proposed one such generic definition. Expected outcome: For this discussion group, I propose to confront this definition with the critique based on other participants' experience and either validate it through examples or give a more appropriate one.

4.5 Compositional Reasoning about Object-Oriented Software Evolution

Einar Broch Johnsen (University of Oslo, NO)

An intrinsic property of real world software is that it needs to evolve. The software is continuously changed during the initial development phase, and existing software may need modifications to meet new requirements. To facilitate the development and maintenance of programs, it is an advantage to have programming environments which allow the developer to alternate between programming and verification tasks in a flexible manner and which ensures correctness of the final program with respect to specified behavioral properties. We propose a formal framework for the flexible development of object-oriented programs, which supports an interleaving of programming and verification steps. The motivation for this framework is to avoid imposing restrictions on the programming steps to facilitate the verification steps, but rather to track unresolved proof obligations and specified properties of a program which evolves. Drawing inspiration from type systems, we use an explicit proof context (or cache) to connect unresolved proof obligations and specified properties, and formulate a soundness in variant for proof contexts which is maintained by both programming and verification steps. The proof context allows a fine-grained analysis of changes to the class hierarchy. Once the set of unresolved obligations in the proof context is empty, the invariant ensures the soundness of the overall program verification.

4.6 Three cases of composition and a question

Ferruccio Damiani (University of Torino, IT)

Three examples of compositional type systems are briefly illustrated. The question is whether people working on programming languages and people workingon formal verification feel the need to identify a suite of code reuse/modularization mechanisms for synergically addressing-fine-grained code reuse- coarse-grained code reuse- spatial/temporal code evolution while being suitable for compositional analysis. Perhaps, being suitable for compositional typing could be a preliminary requisite for such a suite of mechanisms. A reformulation of the question: is it feasible for this research community to agree on a list of recommendations / guidelines / principles to be taken into account when designing a new language (or evolving an existing one) in order to facilitate formal verification?

4.7 Synthesis and Control for Infinite-State Systems with Partial Observability

Rayna Dimitrova (Universität des Saarlandes, DE)

Joint work of Dimitrova, Rayna; Finkbeiner, Bernd

The information available to a component in a distributed system is limited by its interface. Thus, in order to deliver realistic implementations, controller synthesis methods must take into account the fact that the controller has incomplete information about the global state of the system. Incomplete information is a major challenge already for finite-state systems where it makes the synthesis problem exponentially harder. For infinite-state systems the problem is in general undecidable. In particular, for real-time systems the controller synthesis problem becomes undecidable in the presence of incomplete information. We will present a novel approach to timed controller synthesis with safety requirements under incomplete information. We developed the first counterexample-guided abstraction refinement scheme that addresses the two dimensions of complexity – incomplete information and the infinite-statespace. The key innovation of our approach is the automatic synthesis of the observation predicates that are tracked by the controller. Previous methods required these predicates to be given manually. Our procedure relies on abstract counterexamples to guide the search for observations that suffice for controllability. We will outline the techniques for refining the set of observation predicates based on symbolic characterization of spurious counterexamples. We will present experimental results demonstrating better performance than approaches based on brute-force enumeration of observation sets in cases when fine granularity of the observations is necessary.

4.8 Compositional Synthesis of Distributed Systems

Bernd Finkbeiner (Universität des Saarlandes, DE)

In this talk I will illustrate how a logical representation of the synthesis problem for distributed systems facilitates a compositional synthesis approach. I will present a compositional proof rule for Extended Coordination Logic (ECL), a new temporal logic that reasons about the interplay between behavior and informedness in distributed systems. ECL extends linear-time temporal logic with quantification over strategies under incomplete information. ECL subsumes thegame-based temporal logics, including the alternating-time temporal logics, strategy logic, and game logic, and can express the synthesis problem for distributed systems with arbitrary architectures. While ECL is undecidable in general, the compositional proof rule can be used to reduce a general ECL formula to a set of formulas in the decidable fragment of ECL.

4.9 How Decomposition Enhances Security Analysis

Kathi Fisler (Worcester Polytechnic Institute, US)

Joint work of Fisler, Kathi; Krishnamurthi, Shriram

Factoring security policies out into separate program components enables interesting forms of compositional reasoning about security properties. Policies are expressive yet declarative. Both property-based verification and exhaustive semantic differencing (a property-free formal analysis) are tractable on policies. Moreover, running these analyses on policies can leave simpler residues to verify about the programs that use policies. The lesson for this seminar is that decomposition into modules in different languages can open some interesting avenues for making verification feasible and tractable.

4.10 Compositional Verification of Procedural Programs

Dilian Gurov (KTH – Stockholm, SE)

The talk gives a high-level overview of a line of work that Marieke Huisman and I started in early 2001. We develop a verification method for control-flow based temporal safety properties that uses relativization on local component properties as a means of handling variability in code: not yet available component code, evolving components, as well as components that exist in multiple variants as resulting from software product lines. We develop the theory based on flow graphs, simulation and maximal models, and explain the various difficulties in practically implementing the approach. These include a suitable choice of specification formalisms, plus algorithmic support for flow graph extraction, maximal flow graph construction, and model checking. As an example application area we show how hierarchical variability models of software product lines can be used to drive verification in a devide-and-conquer style, allowing for scalable analysis of large numbers of software products.

4.11 Abstract Symbolic Execution

Reiner Haehnle (TU Darmstadt, DE)

Modern software tends to undergo frequent requirement changes and typically is deployed in many different scenarios. This poses significant challenges to formal software verification, because it is not feasible to verify a software product from scratch after each change. It is essential to perform verificationin a modular fashion instead. The goal must be to reuse not merely software artifacts, but also specification and verification effort. In our setting code reuse is realized by delta-oriented programming, an approach where a core program is gradually transformed by code "deltas" each of which corresponds to a product feature. The delta-oriented paradigm is then extended to contract-based formal specifications and to verification proofs. As a next step towards modular verification we transpose Liskov's behavioural subtyping principle to the delta world. Finally, based on the resulting theory, we perform a syntactic analysis of contract deltas that permits to automatically factor out those parts of a verification proof that stays valid after applying a code delta. This is achieved by a novel verification paradigma called "abstract symbolic execution".

4.12 GCM/ProActive: a distributed component model, its implementation, and its formalisation

Ludovic Henrio (INRIA Sophia Antipolis – Méditerranée, FR)

The main claim of this talk is to show that software components and active objects provide an efficient programming model and verification setting. I present a component model that aims at large scale distributed systems, and is implemented as part of the ProActive library. the component model is named GCM, which stands for Grid Component Model. I present several works using formal methods to prove properties on the component model, or on some applications composed with it. I conclude with a few perspectives including verification of adaptive components, and better adaptation to multicore architectures.

4.13 VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java

Bart Jacobs (KU Leuven, BE)

VeriFast is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic. To enable rich specifications, the programmer may define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract separation logic predicates. To enable verification of these rich specifications, the programmer may write lemma functions, i.e., functions that serve only as proofs that their precondition implies their postcondition. The verifier checks that lemma functions terminate and do not have side-effects. Verification proceeds bysymbolic execution, where the heap is represented as a separation logic formula. Since neither VeriFast itself nor the underlying SMT solver do any significant search, verification time is predictable and low. We have used VeriFast to verify fine-grained concurrent data structures, unloadable kernel modules, JavaCard programs, and a network routing program embedded in a home gateway. In this talk, we demonstrate the features of VeriFast for C and Java in a 60-minute tutorial.

4.14 A Variability-Aware Module System

Christian Kaestner (Carnegie Mellon University – Pittsburgh, US)

Module systems enable a divide and conquer strategy to software development. To implement compile-time variability in software product lines, modules can be composed in different combinations. However, this way, variability dictates a dominant decomposition. As an alternative, we introduce a variability-aware module system that supports compile-time variability inside a module and its interface. So, each module can be considered a product line that can be type checked in isolation. Variability can crosscut multiple modules. The module system breaks with the antimodular tradition of a global variability model in product-line development and provides a path toward software ecosystems and product lines of product lines developed in an open fashion. We discuss the design and implementation of such a module system on a core calculus and provide an implementation for C as part of the TypeChef project. Our implementation supports variability inside modules from #ifdef preprocessor directives and variable linking at the composition level. With our implementation, we type check all configurations of all modules of the open source product line Busybox with 811 compile-time options, perform linker check of all configurations, and report found type and linker errors- without resorting to a brute-force strategy.

4.15 Compositionality for Complex Event Processing and Aspects

Shmuel Katz (Technion – Haifa, IL)

Main reference M. Goldman, E. Katz, S. Katz, "MAVEN: modular aspect verification and interference analysis," Formal Methods in System Design, vol. 37, no. 1, pp. 61–92, 2010.
URL http://dx.doi.org/10.1007/s10703-010-0101-1

Aspects can be viewed as system transformers, and then specified by assume guarantee pairs, where the assumption relates to the system to which the aspect is to be woven, and the guarantee to the resultant system after weaving. The correctness of the aspect relative to this assumption can then be shown by creating a model of the assumption's tableau with the aspect state machine model woven in, and checking whether the gurarantee holds for that model. Interference among aspects can also be defined as whether one aspect disturbs the assumption or the guarantee of another. Similar ideas can be used to specify and verify event detectors and responses for complex event processing.

4.16 Composition on the Web

Shriram Krishnamurthi (Brown University – Providence, US)

The modern Web is full of composition. Some of the most interesting (andterrifying) instances are in the browser itself. Broadly, there are two kinds of composition: between the browser and extensions, and within the page itself. My presentation briefly outlines some of these scenarios, to provide challenging examples of composition scenarios on which people can try out their theoretical ideas.

4.17 Compositional Programming

Oscar M. Nierstrasz (Universität Bern, CH)

This talk surveys the evolution of compositional paradigms in programming fromthe 1950s through to the present day. In particular, we explore the innovations introduced by procedural programming, object-oriented langauges, component-based development, and model-driven engineering.

4.18 Synthesis of Control for Component-based systems using Knowledge

Doron A. Peled (Bar-Ilan University – Ramat-Gan, IL)

In distributed systems, local controllers often need to impose global guarantees. A solution that will not impose additional synchronization may not be feasible due to the lack of ability of one process to know the current situation at another. On the other hand, a completely centralized solution will eliminate all concurrency. A good solution is usually a compromise between these extremes, where synchronization is allowed for in principle, but avoided whenever possible. In a quest for practicable solutions to the distributed control problem, one can constrain the executions of a system based on the pre-calculation of knowledge properties and allow for temporary interprocess synchronization in order to combine the knowledge needed to control the system. This type of control, however, may incur a heavy communication overhead. We introduce the use of simple supervisor processes that accumulate information about processes until sufficient knowledge is collected to allow for safe progression. We combine the knowledge approach with a game theoretic search that prevents progressing to states from which there is no way to guarantee the imposed constraints.

4.19 Compositional verification: A tutorial

Arnd Poetzsch-Heffter (TU Kaiserslautern, DE)

The tutorial starts with the presentation of basic principles of compositional verification and suggest to categorize techniques for compositional verification according to four dimensions:

- What are the components? How are they represented?
- What are the composition mechanisms?
- What properties are addressed? How are they specified?
- What kinds of verification techniques are used?

In the main part, the tutorial takes a closer look at four different settings to discuss important aspects in the huge space of compositional verification:

1. Maximal models for model checking step-synchronous Moore machines
2. Model checking control-flow properties of sequential procedural programs
3. Modular state-based verification of object-oriented programs
4. Assume-guarantee reasoning for communicating processes

(The selection of the four settings was partly done to foster the communication of the seminar participants, and partly reflects my restricted knowledge. It does not provide a fair and balanced representation of the space.)

4.20 Typical Worst-Case Analysis of Real-Time Systems

Sophie Quinton (TU Braunschweig, DE)

We present a new compositional approach providing safe quantitative information about real-time systems. Our method is based on a model to describe sporadic overload and bursts at the input of a system. We show how to derive from such a model safe quantitative information about the response time of each task. Experiments demonstrate the efficiency of this approach on a real-life example.

4.21 Component signatures of networking process components require protocol and role declarations

Johannes Reich (SAP AG – Walldorf, DE)

Although from an implementation perspective executable process components are characterized by their computational function, this is not the way they present themselves from their partners' perspective in their network like interactions. From their partners' perspective, only a projection of the interacting process components becomes visible, which is best described as their role in a protocol. Thus, to support component based design of processes, these process components require the declaration of their role and protocol in their component signature.

4.22 A logical perspective on (finite) software systems and their composition

Johannes Reich (SAP AG – Walldorf, DE)

A system notion is proposed that rests on the mathematical function notion, transforming in one time step some input and internal state values onto some output and internal state values. The question whether a certain computational entity represents a system thereby becomes the question to identify the internal and i/o states together with the system function and the corresponding time step. The effect of system interaction on system composition can be classified as:

1. Parallel processing or strict sequential interaction results in strictly hierarchical super system formation
2. Deterministic bidirectional interactions together with certain consistency conditions result in (recursive) super system formation
3. Nondeterministic bidirectional interactions together with certain consistency conditions results in provably no super system formation
4. further, non-classified relations.

4.23 Compositionality, huh?

Arend Rensink (University of Twente, NL)

In this presentation I have formulated a simple formal framework in which some of the essential aspects of compositionality come together. The most important lesson is that the composition operator should be compatible with a notion of abstraction, or semantics, encapsulating the conceptual understanding of the objects under construction. If this compatibility fails, the common solution is to augment the abstraction by putting more information into it. This kind of augmentation is, in fact, the same kind of step as strengthening the induction hypothesis in an inductive proof. The framework is tested, with varying degrees of success, against a number of cases of composition from different domains, ranging from bisimilarity minimisation to testing and subclassing (seen as a composition operator). Some of these cases nicely illustrate the principle of augmentation; others provide examples where compositionality simply fails to hold.

4.24 Comparing Verification Condition Generation with Symbolic Execution

Malte Schwerhoff (ETH Zürich, CH)

Joint work of Schwerhoff, Malte; Kassios, Ioannis T.; Müller, Peter

There are two dominant approaches for the construction of automatic program verifiers, Verification Condition Generation (VCG) and Symbolic Execution (SE). Both techniques have been used to develop powerful program verifiers. However, to the best of our knowledge, no systematic experiment has been conducted to compare them – until now. We have used the specification and programming language Chalice and compared the performance of its standard VCG verifier with a newer SE engine called Syxc, using the Chalice test suite as a benchmark. We have focused on comparing the efficiency of the two approaches, choosing suitable metrics for that purpose. Our metrics also suggest conclusions about the predictability of the performance. Our results show that verification via SE is roughly twice as fast as via VCG. It requires only a small fraction of the quantifier instantiations that are performed in the VCG-based verification.

4.25 Compositional Verification of Actors

Marjan Sirjani (Reykjavik University, IS)

Rebeca is designed as an imperative actor-based language with the goal of providing an easy to use language for modeling concurrent and distributed systems, with formal verification support. I will explain the language Rebeca and the supporting model checking tools. Abstraction and compositional verification, and state-based reduction techniques including symmetry reduction of Rebeca will be discussed. As an example of a domain specific example, I will show how we used Rebeca for model checking SystemC codes.

4.26 Risk Management meets model checking: compositional analysis of DFTs

Marielle Stoelinga (University of Twente, NL)

Dynamic fault trees (DFTs) are a versatile and common formalism to model and analyze the reliability of computer-based systems. This talk presents a formal semantics of DFTs in terms of input/output interactive Markov chains (I/O-IMCs), which extend continuous-time Markov chains with discrete input, output and internal actions. This semantics provides a rigorous basis for the analysis of DFTs. Our semantics is fully compositional, that is, the semantics of a DFT is expressed in terms of the semantics of its elements (i.e. basic events and gates). This enables an efficient analysis of DFTs through compositional aggregation, which helps to alleviate the state-space explosion problem, and yields a very flexible modeling and analysis framework by incrementally building the DFT state space. We have implemented our methodology by developing a tool, and showed, through a number of case studies, the feasibility of our approach and its effectiveness in reducing the state space.

4.27 Modularity and compositionality in embedded system design: interface synthesis and interface theories

Stavros Tripakis (University of California – Berkeley, US)

Compositional methods, that allow to assemble smaller components into larger systems both efficiently and correctly, are not simply a desirable feature in system design: they are a must for designing large and complex systems. In this talk I will present some recent work on this general theme, motivated by embedded system applications. A key notion is that of "interface" which allows to abstract a component, hiding details while exposing relevant information. I will present methods for automatic bottom-up synthesis of interfaces for hierarchical synchronous and dataflow models, motivated by the need for modular code generation from such models. I will also present two interface theories for the same models. Interface theories can be seen as behavioral type theories. They include the key notion of refinement, which captures substitutability: when can a component be replaced by another one without compromising the properties of the entire system. I will present two interface theories: – synchronous relational interfaces, targeted at synchronous systems and functional properties; – actor interfaces, targeted at dataflow models and performance properties such as throughput or latency.

4.28 Interface Theories: Design Choices

Stavros Tripakis (University of California – Berkeley, US)

Interface theories such as interface automata were introduced by Alfaro and Henzinger in the early 2000s. A key characteristic of these theories is the "asymmetry" of inputs and outputs, and the fact that interfaces are not "input-enabled": they may declare some inputs as illegal. This results in a "demonic" notion of composition and an "alternating" notion of refinement. This talk discusses the design choices that naturally lead to these definitions.

4.29 Compositional Behavioral Modeling with Real Time Games

Andrzej Wasowski (IT University of Copenhagen, DK)

Joint work of Wasowski, Andrzej; Larsen, Kim; Legay, Axel; Nyman, Ulrik; David, Alexandre

The presentation introduces the concept of specification theory (or interface)theory for automata-like models, including the main motivation and ingredient operators. I show how we instantiated this paradigm in the tool ECDAR, which aims at stepwise design and verification of real-time embedded controllers. The tool is based on the semantic model of Timed Games, and its associated symbolic solving algorithms. I will present the main objects of ECDAR's specification theory (implementations, specifications, properties), its transformation operators (conjunction, parallel composition, quotient) and its verification operator (satisfaction, refinement). I focus mostly on examples of structuring models and correctness proofs. Time permitting, I will show patterns that allow Assume/Guarantee style of verification, and combine them to obtain (finite) inductive proofs of correctness using refinements in ECDAR.

5 Working Groups

5.1 Working Group: How can model checking and deductive verification benefit from each other/verification of very large systems

Huisman, Marieke; Sirjani, Marjan

A group of about 15 people with very different backgrounds attended this working group. Initially, the discussion went in many different directions, and the point was raised that work on this had been attempted already 20 years ago. However, it some point it was realised that much of this unclearity was caused by the term deductive verification, which can have many different meanings. Thus it was necessary to clarify the difference between theorem proving and program verification. Theorem proving means that one writes some logical properties in a formal language and uses the theorem prover to verify the properties. Two kinds of theorem provers exist: interactive theorem provers, typically used for proving properties in higher-order logic, where the user has to guide the verification process, and automated theorem provers, typically used for first-order logic, where an algorithm tries to construct a proof. Some well-known examples of interactive theorem provers are: HOL, PVS, Isabelle and Coq. Some well-known examples of automated theorem provers are all STMLib-compliants tools, such as Vampire and Z3. Program verification is really focused on the verification of annotated software. Their formal foundations are for example Hoare logic, weakest precondition calculus or separation logic. Typically, they use verification condition generation or symbolic execution to generate proof obligations in first-order logic. To verify these proof obligations, automated theorem proving is used. Then the discussion continued about how program verification and model checking could benefit from each other. It was observed that model checking is currently moving from concurrent models to concurrent software, whereas program verification is moving from sequential software to concurrent software. Thus, the verification targets seem to meet. However, program verification typically focuses on data-oriented properties, whereas model checking focuses on control-flow-oriented properties. Finally, we identified several interesting examples. Program verification could benefit from model checking in the following examples:

- a distributed computation: a server sends a task to a worker. If the task is too big, the worker spawns a new thread to perform part of the task (and this is repeat until the task is small enough). Eventually all the results from the task are merged. This control pattern is difficult to capture with classical program verification techniques;
- verification of e-voting software: typically security properties such as absence of information leakage depend on control and data;
- counter example generation. Model checking could also benefit from program verification, for example in the following case: – control flow properties of code with complicated data structures: suppose we have a device driver that uses a balanced search tree as internal data structure.

The tree would have a method to compute its size. Program verification can be used to prove that the size does not change when for example rebalancing the tree, and the information that the size of the tree is not changed between two consecutive calls can then be used by the model checker to reduce the state space. A next step to continue the results from this workgroup would be to concretely start working on these examples.

5.2 Working Group: Compositional Synthesis of Reactive Systems

Barbara Jobstmann

The working group consisted of four people, all with a similar background in reactive synthesis and automata-based game theory. Therefore, there was no need to discuss the precise meaning of the problem and we could dive right into the technical details about the subject. We brainstormed on the different approaches we knew and explained to each other their key aspects. More precisely, Marielle mentioned recent work on "Compositional Synthesis of Safety Controllers" by her student Wouter Kuijper. We discuss the benefits of using safety specifications and the difficulties arising with more general specifications. Barbara mentioned that similar issues also arise in a problem of aiming to synthesis missing environment assumptions. Here the idea is the given an unrealizable specification we aim for a minimal assumption that makes the specification realizable. Finally, Ufuk discussed his recent work on semi-compositional version of the generalized reactivity-1 approach. Barbara and Rayna provided feedback and suggested extensions. Over all the discussion was very pleased, focused, and technical and more than worth the short amount of time we spent on it.

5.3 Working Group: Modular Full Functional Specification and Verification of C and Java programs that Perform I/O

Bart Jacobs

We discussed preliminary ideas on how to verify I/O properties of C and Java programs. For concreteness, we used the VeriFast program verification tool to illustrate the ideas through simple example programs. We started with a simple C program that prints "Hi" using two calls of the 'putchar' function. Our first specification for this program used an abstract separation logic predicate 'world' with an single parameter of type 'list of character' that represents the characters that have been printed. The precondition stated 'world([])' and the postcondition stated 'world(['H', 'i'])'. This specification approach works fine if we also check termination. Otherwise, the program can circumvent it by first printing e.g. "Bye" and then going into an infinite loop, so that the postcondition is never checked. The second specification used the same 'world' predicate, but now the parameter denotes the characters that are yet to be printed by the program. The precondition states 'world(['H', 'i'])' and the postcondition states 'world([])'. This specification properly enforces the safety property that the program only ever prints a prefix of "Hi", even if it does not terminate. However, this specification approach cannot accommodate nondeterministic behavior. Therefore, we next came up with a predicate 'world' with two parameters: one of type 'list of character', denoting the characters that have been printed, and one of type 'set of lists of characters', denoting the set of allowed values of the first argument. The precondition of 'putchar' requires that the old value of the list, with the new character appended to the end, is in the set. An alternative approach is to have a single argument of type 'iospec', where 'iospec' is coinductively defined as 'function from I/O actions to I/O results' and 'I/O result' is defined as either 'Not Allowed' or 'Allowed (iospec)' which specifies the new I/O specification that holds after the action is performed. We also discussed if it would be possible (and a good idea) to build on this approach to also check liveness properties, by adding 'tau' actions (a.k.a.stutter steps or silent actions). We remarked that it would not in general be easy to modularly prove conformance of a program against such a specification. We noted that it was important to record all actions in a single stream; otherwise, it would not be possible to specify an ordering between the various types of actions. Conclusions: We have some ideas on how to specify and verify I/O properties of Cand Java programs. However, we need to check their feasibility by trying them out on larger examples. We anticipate that additional mechanisms will be necessary to achieve a truly modular approach.

Participants: Erik Poll, Dilian Gurov (first half), Einar Broch Jonsen, Malte Schwerhoff, Wolfgang Ahrendt, Bart Jacobs .

5.4 Working Group: Benchmark for Industrial Verification/Synthesis Problem

Johannes Reich

The interest in the working group o was pretty high. Johannes Reich gave a short introduction (see slides) about potentially interesting problems from an industry perspective. Then we discussed the matter along the following lines: There was general agreement that a benchmark would be an appropriate tool to gain attraction and attention for the advanced engineering methods commonly termed "formal methods" The main issue is to find a showcase that is immediately relevant for the industry like SAP and which provides provides a scalable and significant problem. One example would be the RSA-challenge of prime number factorization, with its immediate relevance for asymmetric encryption techniques. The examples of the introduction were seen as too unspecific. In general, the area of formal methods does not come with _the_ problem but instead shows quite some heterogeneity. In some sense there seems to be a henn-and-egg problem to bring together the more technical view of the formal methods experts and the business view of the industry. An other approach could be that a company like SAP provides an example application where the power of several formal methods like verification, synthesis, etc. could be demonstrated – possibly to arrive at a show case for a benchmark. Interesting candidates for SAP could be the tax engine or the pricing engine, where correctness is top priority and errors might get a high visibility. Another area could be product security or areas where high testing efforts could be ameliorated by formal verification. Thanks for the feedback and best regards to all participants!

Further sources and competitions

Several other competitions and one article was mentioned by the participants:

References

1 M Huisman, V Klebanov and R Monahan (2011) On the Organisation of Program Verification Competitions, http://ceur-ws.org/Vol-873/papers/paper_2.pdf
2 Transformation Tool Competition: http://planet-sl.org/ttc2011 for the last edition, there is a next one coming up for 2013
3 Knowledge Engineering for Planning and Scheduling: http://icaps12.poli.usp.br/icaps12/ickeps
4 Microsoft Research: 2012 Verified Software Milestone Award, http://dream.inf.ed.ac.uk/vsi/

5.5 Working Group: Compositional Verification of Software Product Lines

Ina Schaefer

A software product line is a set of software systems developed from a common set of core assets. There exist several implementation techniques for product lines, such as annotative, compositional or transformational approach, and different analysis strategies which apply to all possible verification approaches. The product-based analysis strategy generates each possible product and verifies it in isolation. The advantage of this strategy is that single-product analyses can be applied without any change, but for large product lines this approach is inefficient or even infeasible. The family-based approach generates a meta-representation of all products which can be analyzed at once in order to derive properties of all products. While this is more efficient than the product-based approach, it uses a closed-world assumption making it fragile to product line evolution. The feature-based analysis strategy takes acompositional approach by analyzing the building blocks of the productsseparately. However, in most cases, this analysis step is not sufficient because it does not take the dependencies and interactions between the building blocks into account. Hence, in addition to the feature-based analysis step another product-based or family-based analysis step is required. In the working group, we discussed the potential of the feature-based compositional analysis strategy for product lines and its limitations. In general, we came to the conclusion that the applicability of this strategy depends on the considered use case. In particular, to assess the above question one needs to fix the implementation approach for the products, the properties that should be verified, the specification technique for the product line, and the analysis technique. It seems that feature-based analysis is in particular well suited to model-checking where traditional assume-guarantee reasoning is adapted as follows: If a feature satisfies its specification under the assumption that it is added to a product with a particular required property, then the resulting composed product guarantees the specification of the feature if there are no interactions between the already contained features and the newly composed one.

6 Programme day-by-day

Monday December 17

9:00 9:05	Welcome from the organisers Tutorial on compositional programming Oscar Nierstrasz
10:35	Coffee
11:00	Personal introductions
12:15	Lunch
13:45 15:15	Tutorial on compositional verification Arnd Poetzsch-Heffter Personal introductions
15:30	Coffee
16:00 17:30	Tutorial on compositional modelling Arend Rensink Remaining personal introductions

Tuesday December 18

9:00	Einar Broch Johnsen Shriram Krishnamurthi Compositionality for web services
10:30	Coffee
11:00	Doron Peled Christian Kaestner A Variability-Aware Module System Wolfgang Ahrendt Compositional semantics and of concurrent/distributed objects
12:15	Lunch
13:45	Sophie Quinton Typical worst-case analysis of real-time systems
14:45	Working groups
17:00	Shmuel Katz Compositionality for Complex Event Processing and Aspects Bernd Finkbeiner Compositional synthesis of distributed systems

Wednesday December 19

9:00	Stavros Tripakis Compositionality and modularity: interfaces everywhere! Wolfgang Ahrendt Compositional verification of concurrent/distributed objects Johannes Reich A logical perspective on software systems and their composition
10:30	coffee
11:00	Ferruccio Damiani Dynamic delta-oriented software product lines Dilian Gurov Compositional Verification of Programs with Procedures Stavros Tripakis design choices for interface theories
12:15	lunch
14:00 14:30	photo excursion

Thursday December 20

9:00	Bart Jacobs VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java
10:00	Reiner Haehnle Abstract Symbolic Execution
10:30	coffee
11:00	Rayna Dimitrova Synthesis and Control for Infinite-State Systems with Partial Observability Malte Schwerhoff Comparing Verification Condition Generation with Symbolic Execution Johannes Reich Component signatures of networking process components require protocol and role declarations
12:15	Lunch
13:45	Marjan Sirjani Compositional Verification of Actors (30 min) Simon Bliudze Glue Synthesis in BIP (30 min.)
14:45	working groups
17:00	Ludovic Henrio GCM/ProActive: a distributed component model, its implement-ation, and its formalisation

Friday December 21

9:15	Mariëlle Stoelinga
9:30	Andrzej Wasowski Compositional design with stepwise refinement using real time games
10:00	Christian Kaestner
10:30	Coffee
11:00	Reports from working groups and plenary discussion
12:00	Closing of the seminar
12:15	Lunch

Participants

- Wolfgang Ahrendt
Chalmers UT – Göteborg, SE
- Simon Bliudze
EPFL – Lausanne, CH
- Einar Broch Johnsen
University of Oslo, NO
- Ferruccio Damiani
University of Torino, IT
- Rayna Dimitrova
Universität des Saarlandes, DE
- Christian Eisentraut
Universität des Saarlandes, DE
- Bernd Finkbeiner
Universität des Saarlandes, DE
- Kathi Fisler
Worcester Polytechnic Inst., US
- Susanne Graf
VERIMAG – Gières, FR
- Dilian Gurov
KTH – Stockholm, SE
- Reiner Hähnle
TU Darmstadt, DE
- Ludovic Henrio
INRIA Sophia Antipolis – Méditerranée, FR
- Marieke Huisman
University of Twente, NL
- Bart Jacobs
KU Leuven, BE
- Barbara Jobstmann
VERIMAG – Gières, FR
- Christian Kästner
Carnegie Mellon University – Pittsburgh, US
- Shmuel Katz
Technion – Haifa, IL
- Shriram Krishnamurthi
Brown Univ. – Providence, US
- Malte Lochau
TU Darmstadt, DE
- Oscar M. Nierstrasz
Universität Bern, CH
- Doron A. Peled
Bar-Ilan University – Ramat-Gan, IL
- Arnd Poetzsch-Heffter
TU Kaiserslautern, DE
- Erik Poll
Radboud Univ. Nijmegen, NL
- Sophie Quinton
TU Braunschweig, DE
- Johannes Reich
SAP AG – Walldorf, DE
- Arend Rensink
University of Twente, NL
- Ina Schaefer
TU Braunschweig, DE
- Malte Schwerhoff
ETH Zürich, CH
- Vasiliki Sfyrla
VISEO – Lyon, FR
- Marjan Sirjani
Reykjavik University, IS
- Lei Song
Universität des Saarlandes, DE
- Martin Steffen
University of Oslo, NO
- Marielle Stoelinga
University of Twente, NL
- Ufuk Topcu
University of Pennsylvania, US
- Stavros Tripakis
University of California – Berkeley, US
- Andrzej Wasowski
IT Univ. of Copenhagen, DK