Multiparty Compatibility for Concurrent Objects

Roly Perera
University of Glasgow, UK
School of Computing Science
roly.perera@glasgow.ac.uk

Julien Lange
Imperial College London
Department of Computing
j.lange@imperial.ac.uk

Simon J. Gay
University of Glasgow, UK
School of Computing Science
simon.gay@glasgow.ac.uk

Objects and actors are communicating state machines, offering and consuming different services at different points in their lifecycle. Two complementary challenges arise when programming such systems. When objects interact, their state machines must be "compatible", so that services are requested only when they are available. Dually, when objects refine other objects, their state machines must be "compliant", so that services are honoured whenever they are promised.

In this paper we show how the idea of *multiparty compatibility* from the session types literature can be applied to both of these problems. We present an untyped language in which concurrent objects are checked automatically for compatibility and compliance. For simple objects, checking can be exhaustive and has the feel of a type system. More complex objects can be partially validated via test cases, leading to a methodology closer to continuous testing. Our proof-of-concept implementation is limited in some important respects, but demonstrates the potential value of the approach and the relationship to existing software development practices.

1 Objects as communicating automata

Two significant state-related challenges arise when programming object-oriented systems. We describe how these both relate to the idea of an object as an automaton, and then present a tool which speaks to both challenges and aligns well with test-driven development practices.

Compatible usage. To use an object safely, one must understand its inner automaton: its states, the services available in each state, and the state transitions that result from service requests. We call this the *compatibility problem*. In standard OO languages the automaton is hidden away behind a flat collection of methods comprising the object's interface. The compatibility problem is compounded by the fact that interacting with one object may indirectly cause a state change in another object.

Compliant refinement. The second challenge is complementary. To safely specialise the behaviour of an object, one must take care to respect its automaton: each state of the overriding object must be contravariant with respect to services offered, and covariant with respect to services consumed. We call this the *compliance problem*. Compliance is the key to robust compositionality: compliant implementations can be safely substituted for abstractions, as embodied by the slogan "require no more, promise no less" [20], allowing the abstractions to serve as boundaries between components. Yet in traditional OO languages compliance is captured at the level of method signatures, not at the level of the object's underlying automaton.

Test-oriented methodologies. The modern context for both of these problems is an increasingly incremental and test-driven development process. In test-driven development [2], programmers deliver features by first defining test cases which characterise them, and then writing enough code to make the tests pass.

D. Orchard and N. Yoshida (Eds.): Programming Language Approaches to Concurrency- and Communication-Centric Software (PLACES 2016).
EPTCS 211, 2016, pp. 73–82, doi:10.4204/EPTCS.211.8

© R. Perera, J. Lange & S. J. Gay
This work is licensed under the
Creative Commons Attribution License.

Having a suite of tests makes it easier to refactor code and increases the visibility of changes to observable behaviour.

Test-driven development has evolved into a host of sophisticated techniques and tools. *Mock objects* [14] are a way of simulating, in the test environment, the other participants in a multi-agent system. *Continuous testing* [23, 21] is a tool feature that automatically runs tests in the background to shorten the feedback cycle between code changes and test failures. These and similar practices are the context in which we wish to propose a new approach to the compatibility and compliance problems.

1.1 Contributions of paper

Our proposal is an approach to language implementation which integrates compatibility checking and compliance checking of automata directly into the programming environment (§ 2). This somewhat blurs the distinction between language and tool: an interactive, incremental style of programming, with immediate feedback on failures, is baked into the way the language works. The distinguishing feature of our approach is that there are no types: compatibility and compliance checking is at the level of objects, making the approach a form of language-integrated continuous testing. Any object or system of objects may serve both as an abstraction and as a prototypical implementation.

Our goal in this paper is to motivate our language/tool design and the development methodology it supports (§ 2). In § 3 we outline the theoretical developments that will be needed to put the approach on a sound footing, including possible extensions and modifications to the notions of communicating automata [8] and multiparty compatibility [11, 12, 5, 9, 18] from the theory of session types. We also mention connections to established techniques such as model checking.

2 Language-integrated verification

Actors [1, 17] are the natural setting for exploring our proposed tool design, since we are concerned with an object's "inner automaton". Unlike other OO paradigms actors make this automaton explicit, allowing the object to change its interface at runtime.

We start with a fairly standard actor calculus and then generalise in some respects and restrict in others. An asynchronous send to an actor `p` is non-standard in allowing one of a set of messages to be chosen non-deterministically, and is written $\mathtt{p}\blacktriangleleft\{\mathtt{m}_1(\mathtt{v}_1)\mathtt{P}_1 \ldots \mathtt{m}_n(\mathtt{v}_n)\mathtt{P}_n\}$. Non-determinism allows objects to be general enough to serve as specifications, but is also convenient for testing. A blocking receive which waits until one of a set of possible messages arrives from `p` is written $\mathtt{p}\blacktriangleright\{\mathtt{m}_1(\mathtt{x}_1)\mathtt{P}_1 \ldots \mathtt{m}_n(\mathtt{x}_n)\mathtt{P}_n\}$, and binds the parameter `x` to the value received. An unrealistically strong but expedient assumption for now is that objects cannot be allocated dynamically but have a fixed configuration. Finally, an actor maintains a unidirectional FIFO queue per client object.

Our implementation is only intended as a proof-of-concept and as such is unable to produce standalone code. A hosted version will be made available online before the workshop.

Compatible usage. Our tool's support for compatibility testing is illustrated on the left-hand side of Figure 1. The code models a simple software development workflow with four objects, of which only two, `teamLead` and `devTeam`, are defined here.

The team lead iterates through a release cycle. At each cycle, the team lead waits to receive a release candidate from the dev team, which the business must then evaluate. The business responds either by instructing the team lead to iterate again, or by accepting the release. In the first case, the team lead tags

```
system dev

obj teamLead
behaviour ReleaseCycle
   devTeam▸releaseCandidate
   business◂evaluate
   business▸{
      iterate(tag)
         repository◂tagRC(tag)
         devTeam◂continue
         ReleaseCycle
      accept(tag)
         repository◂tagRelease(tag)
         devTeam◂stop.
   }
ReleaseCycle

obj devTeam
repository◂commit
repository▸revision(n)
behaviour ReleaseCycle
   teamLead◂releaseCandidate
   teamLead▸{
      continue
         repository◂commit
         repository▸revision(n)
         ReleaseCycle
   }
ReleaseCycle
```

```
system dev-refactored: dev

obj teamLead
behaviour ReleaseCycle
   devTeam▸releaseCandidate
   business◂evaluate
   business▸{
      accept(tag)
         repository◂tagRC(tag)
         devTeam◂stop.
   }
ReleaseCycle

obj devTeam
behaviour ReleaseCycle
   repository◂commit
   repository▸revision(n)
   teamLead◂releaseCandidate
   teamLead▸{
      continue
         ReleaseCycle
      stop.
   }
ReleaseCycle
```

Figure 1: Partially refactored `dev` system, with two implementation errors

the release candidate in the repository and repeats, having told the dev team to continue. Otherwise, the release candidate is tagged as an official release in the repository and the dev team instructed to stop. The dev team has its own notion of release cycle; on each iteration it commits code to the repository, notifies the team lead, and then awaits an instruction to continue.

Our tool highlights two compatibility errors using wavy underlining. It has detected a state in which the message `stop` message sent to the dev team will never be delivered, and another state in which the `continue` message that the team lead is waiting for will never arrive. (The colour of the underlining reflects the polarity of the message – either sending or receiving – but is technically redundant given the arrows on the messages.) An important detail to notice here is that this system is not closed: the objects *`business`* and *`repository`* are unspecified external agents (indicated by the italicised names).

Compliant refinement. Compliance testing is also illustrated in Figure 1. On the right-hand side of the figure is `dev-refactored`, an attempt to refine the observable behaviour of `dev`. The programmer specifies this intention using the colon syntax in the `system` definition. The colon is analogous to Java `implements` rather than Java `extends`, as no implementation is inherited from `dev`.

Our tool highlights two compliance errors using solid underlining. First, the programmer has not implemented `iterate`. If `dev-refactored` were substituted for `dev`, it would be unable to satisfy its contract. This is analogous to failing to implement a Java interface method, except that in this language the interface (set of available methods) varies explicitly from state to state. Although the system which is at fault is `dev-refactored`, the error is indicated by underlining the unmet obligation in `dev`, namely the `iterate` handler. (The errors should be understood as a view contextualised to `dev-refactored`.)

The second error is in the new implementation of `accept`, where the programmer has called `tagRC` rather than `tagRelease`. If `dev-refactored` were substituted for `dev`, it would require services of its environment that might not be on offer. This error has no analogue in Java's type system, since it corresponds to checking that the *implementation* of an overriding method refines the observable behaviour of the overridden method.

Moreover, the programmer was able to consolidate the duplicate `commit`/`revision` interaction with the repository in the original dev team code into a single interaction at the beginning of the loop in the new code. Our implementation is able to verify that this is a behaviour-preserving change. This emphasises how in our language, behavioural specifications are *prototypical implementations* that can form part of an actual system, or be refined into more specialised implementations. The design pattern *template method* [15] is similar in intent but requires the programmer to decide in advance which aspects of the behaviour are fixed and which can be overridden by a refining implementation.

Test-oriented methodologies. The compatibility and compliance features just described are related to *continuous testing* [23, 21], in that testing is automatic and feedback is immediate. However our approach provides more automation than continuous testing tools, which automate test execution but still require the programmer to write the tests in the first place. In our implementation all possible interactions between simple finite-state objects are verified automatically without the programmer having to write explicit tests.

```
system repo

obj repository
behaviour Connected(n)
   devTeam▸commit
   devTeam◂revision(n)
   teamLead▸{
      tagRC(tag)
         math◂plus(n, 1)
         math▸val(m)
         Connected(m)
      tagRelease(tag).
   }
Connected(0)
```

```
system repo-test
using repo
using dev

obj business
teamLead▸evaluate
teamLead◂{
   accept("1.0").
   iterate("1.0RC")
      teamLead▸evaluate
      teamLead◂accept("1.0").
}
```

Figure 2: Non-finite state `repository`, with `business` test case

Of course exhaustively verifying even a finite-state system quickly becomes impractical as its state-space grows. While techniques from model checking [13, 22, 19] and symbolic execution [7] may be applicable, this problem can also be addressed by adopting a more manual test-oriented style. The system to be tested is placed into an environment which has been simplified by replacing some components by

mock objects [14] so that the state-space of the resulting system is small enough for exhaustive checking to be feasible.

This is illustrated in Figure 2 on the previous page. The repository is the code to be tested; its states are indexed with a natural number incremented on every `commit`, representing the current revision; there is therefore a state for every natural number. The verification of this object is made tractable by composing it with a simplified environment: in this case a mock business object which hard-codes a representative sequence of interactions. (Other components are imported from `dev` by the `using` declaration.) The resulting system has a small number of states which our tool can automatically and exhaustively verify. Like testing in general, this method is complete (all reported errors are *bona fide* errors) but not sound with respect to the larger system being tested, which may contain problems not revealed by the test.

```
obj teamLead
behaviour ReleaseCycle
   devTeam▸releaseCandidate
   business◂evaluate
   business▸{
      iterate(tag)
         repository◂tagRC(tag)
         devTeam◂continue
         ReleaseCycle
      accept(tag)
         repository◂tagRelease(tag)
         devTeam◂stop.
      discard
         repository◂revert
         devTeam◂continue
         ReleaseCycle
   }
ReleaseCycle
```

```
obj repository
behaviour Connected(n)
   devTeam▸commit
   devTeam◂revision(n)
   teamLead▸{
      tagRC(tag)
         math◂plus(n, 1)
         math▸val(m)
         Connected(m)
      tagRelease(tag).
   }
Connected(0)

obj business
teamLead▸evaluate
teamLead◂{
   accept("1.0").
   iterate("1.0RC")
      teamLead▸evaluate
      teamLead◂{
         accept("1.0").
         discard.
      }
}
```

Figure 3: New `discard` service (left) verified by adding branch to test case (centre)

Notice how non-deterministic choice substantially reduces the coding overhead usually associated with testing. In Figure 3 the programmer implements a new service on the team lead, giving the business the option to discard the release candidate (left, green highlight); if this happens the repository is reverted to the previous revision, and the dev team told to start a new iteration. Rather than writing a whole new test case, the programmer can simply add a `discard` branch to the existing test case (right, green highlight). This immediately yields errors on `tagRC` and `tagRelease`, and the complementary error on `revert`, reflecting the pending obligation to add `revert` as a service to the repository.

3 Extensions to communicating automata and multiparty compatibility

Several extensions to the theory of communicating automata and multiparty compatibility will be required to formalise our proposed approach. We outline some of the basic ideas here, assuming some familarity with the usual definitions. The extensions we anticipate are compositional multiparty compatibility (§ 3.1), dynamic allocation of objects (§ 3.2) and communicating automata that exchange values (§ 3.3).

3.1 Compositional multiparty compatibility

The first requirement is to extend multiparty compatibility to systems of objects where not all the actors are known. Here we sketch the approach used in our implementation. First we define a notion of composition of automata that distinguishes between "internal" and "external" transitions, illustrated here by example. In Figure 4 the basic automata C and D have p and q respectively as their subjects. A message between p and q is therefore *internal* to the composite $C \otimes D$, since both participants belong to the composite automaton. A message is *external* iff it is not internal.

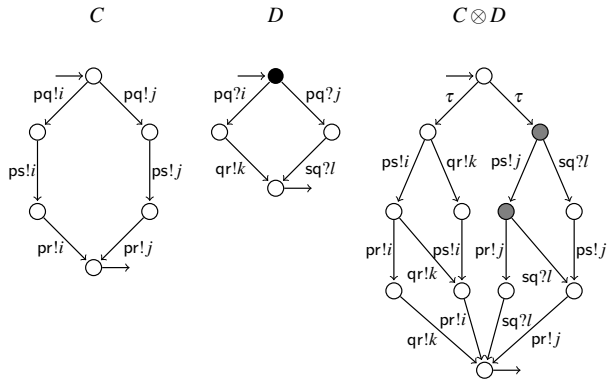

Figure 4: Composition of basic automata, distinguishing internal and external actions

For brevity internal actions are shown as τ in the figure but formally they are synchronous transitions of the form $\mathsf{p} \to \mathsf{q} : i(v)$. In the context of $C \otimes D$ it is then meaningful to ask whether C and D are *compatible*, written $C \bowtie D$. For this we use a definition of multiparty compatibility in the style of [4], except that we only require that *internal* interactions of C and D always have a potential rendezvous partner. In Figure 4 the automata are compatible; the initial states of C and D are the only ones that need checking because the others involve only external transitions.

In Figure 5, we compose the composite automaton $C \otimes D$ with another basic automaton C', which has r as its subject.

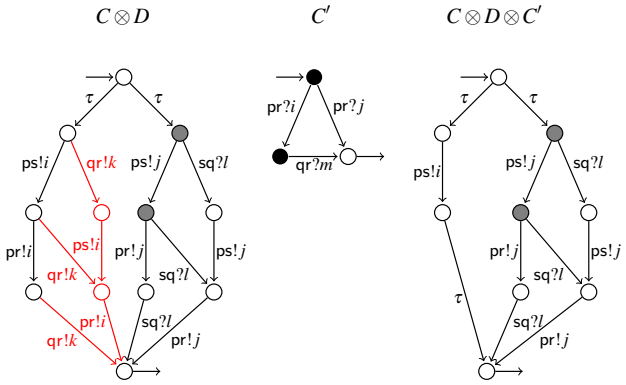

Figure 5: Incompatibility of $C \otimes D$ with C', assuming $k \neq m$

Now the internal transitions that need verifying are those between p and r or between q and r. Assuming $k \neq m$, the transitions highlighted in red are unable to synchronise and so are dropped from the composite $C \otimes D \otimes C'$. A send which cannot synchronise (or dually, a set of receives, none of which can synchronise) does not *ipso facto* represent an incompatibility, since it may synchronise in a later state. However in this case the message qr!k can never be delivered, and so $C \otimes D$ and C' are incompatible. This incompatibility is preserved by any inclusion of $C \otimes D \otimes C'$ into a larger system.

3.1.1 Safety properties of compatibility

There are two key safety properties that compatibility must guarantee. The first is a variant of the usual property that executing C and D in parallel under an asynchronous semantics never leads to a deadlocked or orphan-message configuration. The variation we need is that the asynchronous semantics must be defined for systems where are not all parties are present in the configuration. As with $\otimes$, we distinguish between internal and external transitions, and then use this to define an asynchronous semantics which only uses queues for internal transitions. For example for Figure 4, the configuration would contain queues only for the channels pq and qp. External sends disappear into the ether, and external receives never block. How this might work when messages includes values is discussed in § 3.3 below.

Even if C and D are compatible, it is always possible to compose them with other objects that cause the whole system to have bad reachable configurations. But there should be an important completeness property, namely that if C and D are *incompatible* then there are no objects they can be composed with which will render them compatible. Then any incompatibility or non-compliance errors reported to the programmer are genuine, supporting the incremental programming methodology we outlined in § 2.

The second property we require for compatibility is that it be preserved by refinement. In other words, if $C \bowtie D$ and $C' \lesssim C$ and $D' \lesssim D$ then $C' \bowtie D'$ where $\lesssim$ is the refinement preorder (to be defined). In the more test-oriented examples of § 2, the objects being composed do not represent the full set of system behaviours but *test cases* that only exercise some region of the full state-space. The property that refinement preserves compatibility means that any properties validated by a test case are valid for any refinements of those objects, although there will in general be properties of the full system which are not validated by those particular tests.

3.2 Dynamic allocation of objects

Although the toy examples presented earlier did not require it, dynamic object creation is essential for real applications. [6] extend communicating automata with object creation; we will need to extend multiparty compatibility to such systems.

3.3 Communicating automata that exchange values

A fundamental extension of the theories based on multiparty compatibility is required to allow automata to exchange values. The major difficulty in dealing with values is that the state-space of the systems considered may grow exponentially, or worse, for example if an automaton may send unbounded integers. However, we can rely on several existing theories and techniques to ensure enough expressivity at the automata level while keeping the logic dealing with values decidable and manageable. One direction to consider is translation from our automata to processes in a modelling language such as mCRL2 [16] which permits the usage of sophisticated data types and associated predicates. These can then be checked via the corresponding tool chain [10].

On the one hand, we know that the infinite nature of communication through unbounded FIFO channels can be dealt with through multiparty compatibility. On the other hand, potentially infinite data types can be dealt with through model checking theories. However, exchanged values and pure interactions are not always orthogonal. To deal with the effect of data on the communication patterns, we will adopt a conservative strategy such as the one used in [3].

4 Conclusion

We proposed a language and tool design where simple concurrent objects are checked exhaustively for compatibility and compliance, and more complex objects can be partially verified via test cases which are executed automatically. Although several non-trivial problems need to be solved before this can applied to realistic examples, we hope to have demonstrated the potential value of the approach and established some connections with existing software development practices and design patterns.

Acknowledgements. Supported by EPSRC (EP/K034413/1 and EP/L00058X/1) and COST Action IC1201.

References

[1] Gul Agha (1986): *Actors: A model of concurrent computation in distributed systems*. The MIT Press, Cambridge, MA, USA.

[2] Kent Beck (2002): *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Company, Inc., Boston, MA, USA.

[3] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-Contract for Distributed Multiparty Interactions*. In: *Concurrency Theory, 21st International Conference, CONCUR 2010*, pp. 162–176, doi:10.1007/978-3-642-15375-4_12.

[4] Laura Bocchi, Julien Lange & Nobuko Yoshida (2015): *Meeting Deadlines Together*. In Luca Aceto & David de Frutos Escrig, editors: *Concurrency Theory, 26th International Conference, CONCUR 2015*, *Leibniz International Proceedings in Informatics (LIPIcs)* 42, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 283–296, doi:10.4230/LIPIcs.CONCUR.2015.283.

[5] Laura Bocchi, Weizhen Yang & Nobuko Yoshida (2014): *Timed Multiparty Session Types*. In Paolo Baldan & Daniele Gorla, editors: *Concurrency Theory, 25th International Conference, CONCUR 2014*, *Lecture Notes in Computer Science* 8704, Springer Berlin Heidelberg, pp. 419–434, doi:10.1007/978-3-662-44584-6_29.

[6] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara & Thomas Schwentick (2013): *Dynamic Communicating Automata and Branching High-Level MSCs*. In Adrian-Horia Dediu, Carlos Martín-Vide & Bianca Truthe, editors: *Language and Automata Theory and Applications*, *Lecture Notes in Computer Science* 7810, Springer Berlin Heidelberg, pp. 177–189, doi:10.1007/978-3-642-37064-9_17.

[7] Robert S. Boyer, Bernard Elspas & Karl N. Levitt (1975): *SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution*. In: *Proceedings of the International Conference on Reliable Software*, ACM, New York, NY, USA, pp. 234–245, doi:10.1145/800027.808445.

[8] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *Journal of the ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[9] Marco Carbone, Fabrizio Montesi, Nobuko Yoshida & Carsten Schurmann (2015): *Multiparty Session Types as Coherence Proofs*. In: *Concurrency Theory, 26th International Conference, CONCUR 2015*, Leibniz International Proceedings in Informatics, doi:10.4230/LIPIcs.CONCUR.2015.412.

[10] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink & Tim A. C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In: *TACAS 2013*, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.

[11] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In: *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, Springer-Verlag, pp. 194–213, doi:10.1007/978-3-642-28869-2_10.

[12] Pierre-Malo Deniélou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In Fedor V. Fomin, RÅńsiÅĘÅą Freivalds, Marta Kwiatkowska & David Peleg, editors: *Automata, Languages, and Programming*, *LNCS* 7966, Springer Berlin Heidelberg, pp. 174–186, doi:10.1007/978-3-642-39212-2_18.

[13] E. Allen Emerson & Edmund M. Clarke (1980): *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*. In: *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, Springer-Verlag, London, UK, pp. 169–181, doi:10.1007/3-540-10003-2_69.

[14] Steve Freeman & Nat Pryce (2009): *Growing Object-Oriented Software, Guided by Tests*, 1st edition. Addison-Wesley Professional.

[15] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Addison-Wesley, Boston, MA, USA.

[16] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and analysis of communicating systems*. MIT Press.

[17] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 235–245.

[18] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, ACM, New York, NY, USA, pp. 221–232, doi:10.1145/2676726.2676964.

[19] Julien Lange & Nobuko Yoshida (2016): *Characteristic Formulae for Session Types*. In: *TACAS 2016*, *LNCS* 9636, Springer, pp. 833–850, doi:10.1007/978-3-662-49674-9_52.

[20] Barbara Liskov (1987): *Keynote Address - Data Abstraction and Hierarchy*. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, ACM, New York, NY, USA, pp. 17–34, doi:10.1145/62138.62141.

[21] Lech Madeyski & Marcin Kawalerowicz (2013): *Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool*. In: *ENASE 2013 - Proceedings of the 8th*

International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 260–267, doi:10.5220/0004587202600267.

[22] J.P. Queille & J. Sifakis (1982): *Specification and verification of concurrent systems in CESAR*. In Mariangiola Dezani-Ciancaglini & Ugo Montanari, editors: *International Symposium on Programming, Lecture Notes in Computer Science* 137, Springer Berlin Heidelberg, pp. 337–351, doi:10.1007/3-540-11494-7_22.

[23] David Saff & Michael D. Ernst (2004): *Continuous testing in Eclipse*. In: *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, doi:10.1016/j.entcs.2004.02.051.