JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java[⋆]

Simon Bliudze$^{1[0000-0002-7900-5271]}$, Petra van den Bos$^{2[0000-0002-9212-1525]}$, Marieke Huisman$^{2[0000-0003-4467-072X]}$, Robert Rubbens$^{2[0000-0002-5638-5945]}$, and Larisa Safina$^{1[0000-0002-4490-7451]}$

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
`simon.bliudze@inria.fr`, `larisa.safina@inria.fr`

[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
`p.vandenbos@utwente.nl`, `m.huisman@utwente.nl`, `r.b.rubbens@utwente.nl`

Abstract. We present "Verified JavaBIP", a tool set for the verification of JavaBIP models. A JavaBIP model is a Java program where classes are considered as components, their behaviour described by finite state machine and synchronization annotations. While JavaBIP guarantees execution progresses according to the indicated state machines, it does not guarantee properties of the data exchanged between components. It also does not provide verification support to check whether the behaviour of the resulting concurrent program is as (safe as) expected. This paper addresses this by extending the JavaBIP engine with run-time verification support, and by extending the program verifier VerCors to verify JavaBIP models deductively. These two techniques complement each other: feedback from run-time verification allows quicker prototyping of contracts, and deductive verification can reduce the overhead of run-time verification. We demonstrate our approach on the "Solidity Casino" case study, known from the VerifyThis Collaborative Long Term Challenge.

1 Introduction

Modern software systems are inherently concurrent: they consist of multiple components that run simultaneously and share access to resources. Component interaction leads to resource contention, and if not coordinated properly, can compromise safety-critical operations. The concurrent nature of such interactions is the root cause of the sheer complexity of the resulting software [9]. Model-based coordination frameworks such as Reo [5] and BIP [6] address this issue by providing models with a formally defined behaviour and verification tools.

JavaBIP [10] is an open-source Java implementation of the BIP coordination mechanism. It separates the application model into *component behaviour*, modelled as Finite State Machines (FSMs), and *glue*, which defines the possible stateless interactions among components in terms of synchronisation constraints. The overall behaviour of an application is to be enforced at run time

[⋆] L. Safina and S. Bliudze were partially supported by ANR Investissments d'avenir (ANR-16-IDEX-0004 ULNE) and project NARCO (ANR-21-CE48-0011). P. van den Bos, M. Huisman and R. Rubbens were supported by the NWO VICI 639.023.710 Mercedes project.

by the framework's engine. Unlike BIP, JavaBIP does not provide automatic code generation from the provided model; instead it realises the coordination of existing software components in an exogenous manner, relying on component annotations that provide an abstract view of the software under development.

To model component behaviour, methods of a JavaBIP program are annotated with FSM transitions. These annotated methods model the actions of the program components. Computations are assumed to be terminating and non-blocking. Furthermore, side-effects are assumed to be either represented by the change of the FSM state, or to be irrelevant for the system behaviour. Any correctness argument for the system depends on these assumptions. A limitation of JavaBIP is that it does not guarantee that these assumptions hold. This paper proposes a joint extension of JavaBIP and VerCors [11] providing such guarantees about the implementation statically and at run time.

VerCors [11] is a state-of-the-art deductive verification tool for concurrent programs that uses permission-based separation logic [3]. This logic is an extension of Hoare logic that allows specifying properties using contract annotations. These contract annotations include permissions, pre- and postconditions and loop invariants. VerCors automatically verifies programs with contract annotations. To verify JavaBIP models, we (i) extend JavaBIP annotations with verification annotations, and (ii) adapt VerCors to support JavaBIP annotations. VerCors was chosen for integration with JavaBIP because it supports multi-threaded Java, which makes it straightforward to express JavaBIP concepts in its internal representation. To analyze JavaBIP models, VerCors transforms the model with verification annotations into contract annotations, leveraging their structure as specified by the FSM annotations and the glue.

For some programs VerCors requires extra contract annotations. This is generally the case with `while` statements and when recursive methods are used. To enable properties to be analysed when not all necessary annotations are added yet, we extend the JavaBIP engine with support for run-time verification. During a run of the program, the verification annotations are checked for that specific program execution at particular points of interest, such as when a JavaBIP component executes a transition. The run-time verification support is set up in such a way that it ignores any verification annotations that were already statically verified, reducing the overhead of run-time verification.

This paper presents the use of deductive and run-time verification to prove assumptions of JavaBIP models. We make the following contributions:

- We extend regular JavaBIP annotations with pre- and postconditions for transitions and invariants for components and states. This allows checking design assumptions, which are otherwise left implicit and unchecked.
- We extend VerCors to deductively verify a JavaBIP model, taking into account its FSM and glue structure.
- We add support for run-time verification to the JavaBIP engine.
- We link VerCors and the JavaBIP engine such that deductively proven annotations need not be monitored at run-time.
- Finally, we demonstrate our approach on a variant of the Casino case study from the VerifyThis Collaborative Long Term Challenge.

Tool binaries and case study sources are available through the artefact [7].

2 Related Work

There are several approaches to analyse behaviours of abstract models in the literature. Bliudze et al. propose an approach allowing verification of infinite state BIP models in the presence of data transfer between components [8]. Abdellatif et al. used the BIP framework to verify Ethereum smart contracts written in Solidity [1]. Mavridou et al. introduce the VeriSolid framework, which generates Solidity code from verified models [13]. André et al. describe a workflow to analyse Kmelia models [4]. They also describe the COSTOTest tool, which runs tests that interact with the model. Thus, these approaches do not consider verification of model implementation, which is what we do with Verified JavaBIP. Only COSTOTest establishes a connection between the model and implementation, but it does not guarantee memory safety or correctness.

There is also previous work on combining deductive and runtime verification. The following discussion is not exhaustive. Generally, these works do not support concurrent Java and JavaBIP. Nimmer et al. infer invariants with Daikon and check them with ESC/Java [14]. However, they do not check against an abstract model, and the results are not used to optimize execution. Bodden et al. and Stulova et al. optimize run-time checks using static analysis [12,16]. However, Stulova et al. do not support state machines, and Bodden et al. do not support data in state machines. The STARVOORS tool by Ahrendt et al. is comparable to Verified JavaBIP [2]. Some minor differences include the type of state machine used, and how Hoare triples are expressed. The major difference is that it is not trivial to support concurrency in STARVOORS. VerCors and Verified JavaBIP use separation logic, which makes concurrency support straightforward.

3 JavaBIP and Verification Annotations

JavaBIP annotations capture the FSM specification and describe the behaviour of a component. They are attached to classes, methods or method parameters, and were first introduced by Bliudze et al [10]. Listing 1 shows an example of JavaBIP annotations. `@ComponentType` indicates a class is a JavaBIP component and specifies its initial state. In the example this is the `WAITING` state. `@Port` declares a transition label. Method annotations include `@Transition`, `@Guard` and `@Data`. `@Transition` consists of a port name, start and end states, and optionally a guard. The example transition goes from `WAITING` to `PINGED` when the `PING` port is triggered. The transition has no guard so it may always be taken. `@Guard` declares a method which indicates if a transition is enabled. `@Data` either declares a getter method as outgoing data, or a method parameter as incoming data. Note that the example does not specify when ports are activated. This is specified separately from the JavaBIP component as glue [10].

We added component invariants and state predicates to Verified JavaBIP as class annotations. `@Invariant(expr)` indicates `expr` must hold after each component state change. `@StatePredicate(state, expr)` indicates `expr` must hold in state `state`. Pre- and postconditions were also added to the `@Transition` annotation. They have to hold before and after execution of the transition. `@Pure`

```
@Port(name = PING, type = PortType.enforceable)
@ComponentType(initial = WAITING, name = ECHO_SPEC)
public class Echo {
  @Transition(name = PING, source = WAITING, target = PINGED)
  public void ping() { System.out.println(this + ": pong");}}
```

Listing 1. Example of a minimal pinging component in JavaBIP

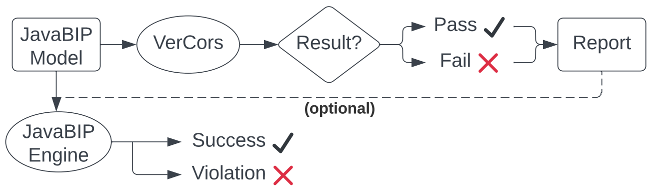

Fig. 1. Verified JavaBIP architecture. Ellipse boxes represent analysis or execution.

indicates that a method is side-effect-free, and is used with `@Guard` and `@Data`. Annotation arguments should follow the grammar of Java expressions. We do not support lambda expressions, method references, switch expressions, `new`, `instanceOf`, and wildcard arguments. In addition, as VerCors does not yet support Java features such as generics and inheritance, models that use these cannot be verified. These limitations might be lifted in the future.

4 Architecture of Verified JavaBIP

The architecture of Verified JavaBIP is shown in Figure 1. The user starts with a JavaBIP model, optionally with verification annotations. The user then has two choices: verify the model with VerCors, or execute it with the JavaBIP Engine.

We extended VerCors to transform the JavaBIP model into the VerCors internal representation, Common Object Language (COL). An example of this transformation is given in Listing 2. If verification succeeds, the JavaBIP model is memory safe, has no data races, and the components respect the properties specified in the verification annotations. In this case, no extra run-time verification is needed. If verification fails, there are either memory safety issues, components violate properties, or the prover timed out. In the first case, the user needs to change the program or annotations and retry verification with VerCors. This is necessary because memory safety properties cannot be checked with the JavaBIP engine, and therefore safe execution of the JavaBIP model is not guaranteed. In the second and third case, VerCors produces a verification report with the verification result for each property.

We extended the JavaBIP engine with run-time verification support. If a verification report is included with the JavaBIP model, the JavaBIP engine uses it to only verify at run-time the verification annotations that were not verified deductively. If no verification report is included, the JavaBIP engine verifies all verification annotations at run time.

```
1  @Transition(name=PING,source=PING,target=PING,guard=HAS_PING)
2  public void ping() { pingsLeft--; }
```

```
1  requires PING_state_predicate() && hasPing();
2  ensures PING_state_predicate();
3  public void ping() { pingsLeft--; }
```

Listing 2. Top: example of a transition in JavaBIP. Bottom: internal representation of `ping` after encoding JavaBIP semantics.

5 Implementation of Verified JavaBIP

This section briefly discusses relevant implementation details for Verified JavaBIP.

Run-time verification in the JavaBIP engine is performed by checking component properties after component construction, and before and after transitions. For example, before the JavaBIP engine executes a transition, it checks the component invariant, the state invariant, and the precondition of the transition. When a property is violated, either execution is terminated or a warning is printed, depending on how the user configured the JavaBIP engine. We expect runtime verification performance to scale linearly, as properties can be checked individually. We have not measured the impact of the use of reflection in the JavaBIP engine.

For deductive verification the JavaBIP semantics is encoded into COL. We describe this with an example. The top part of Listing 2 shows the `ping` method, where `@Transition` indicates a transition from `PING` to `PING`. The guard indicates that the transition is allowed if there is a ping. `HAS_PING` refers to a method annotated with `@Guard(name=HAS_PING)`, which returns `pingsLeft >= 1`.

The bottom part of Listing 2 shows the COL representation of the `ping` method after encoding the JavaBIP semantics. Line 1 states the precondition, line 2 the postcondition. `PING_state_predicate()` refers to the `PING` state predicate, which constrains the values of the class fields. By default it is just `true`. Since the predicate is both a pre- and a postcondition, it is assumed at the start of the method, and needs to hold at the end of the method. `hasPing()` is the method with the `@Guard` annotation for the `HAS_PING` label. The method is called directly in the COL representation. We have implemented such a transformation of JavaBIP to COL for each JavaBIP construct.

To prove memory safety, we extended VerCors to generate permissions. This ensures verification in accordance with the Java memory model. Currently, each component owns the data of all its fields. This works for JavaBIP models that do not share data between components. For other models, a different approach might be necessary, e.g. VerCors taking into account permissions annotations provided by the user. For more info about permissions, we refer the reader to [3].

Finally, scalability of deductive verification of JavaBIP models could be a point of future work, as the number of proof obligations scales quadratically in the number of candidate transitions of a synchronization.

6 VerifyThis Casino and Verified JavaBIP

We illustrate Verified JavaBIP with the Casino case study adapted from [17]. We discuss the case study and its verification. The case study sources and Verified JavaBIP sources and binaries are included in the artefact [7].

The model uses three component types: player, operator, and casino. The model supports multiple players and casinos, but each casino has only one operator. Players bet on the result of a coin flip. The casino pays out twice for a correct guess, and keeps the money otherwise. The casino contains the pot balance and money reserved for the current bet. The operator can add to or withdraw money from the casino pot based on a local copy of the casino pot.

We have added several invariants to this model. The purse of every player, the casino pot, its operator copy, the wallet of the operator, and the placed bet must all be non-negative, as the model does not support debts. If no bet is placed, it must be zero. These properties are defined as `@Invariant` or `@StatePredicate` annotations on the components in the model (see **??**).

One problem with the model is that the player can win more than the casino pot contains, because there are no restrictions on how much the player can bet. The problem is detected by both deductive and run-time verification. VerCors cannot prove that the casino pot is non-negative, which is part of the casino invariant, after the `PLAYER_WIN` transition. The JavaBIP engine detects it, but is not guaranteed to because the model has some non-determinism. For example, if no player ever wins the problem is not detected by run-time verification.

There are several solutions. First, the user can choose to always enable run-time verification, such that the execution is always safe. This might be acceptable depending on the performance penalty of run-time verification. Second, guards can be added to restrict model behaviour. For example, `PLACE_BET` could require `bet <= pot`. However, in general, adding guards might introduce deadlocks. Third, a solution is to refactor the model to avoid the problem. For example, the casino could limit how much the player can bet. This introduces no extra run-time checks, however, in general the behaviour of the model will change.

7 Conclusions and Future Work

We presented Verified JavaBIP, a tool set for verifying the assumptions of JavaBIP models and their implementations. The tool set extends the original JavaBIP annotations for verification of functional properties. Verified JavaBIP supports deductive verification using VerCors, and run-time verification using the JavaBIP engine. Only properties that could not be verified deductively are checked at runtime. In the demonstration we automatically detect a problem on the Casino case study using Verified JavaBIP.

There are several directions for future work. First, support for checking memory safety could be extended by supporting data sharing between components. Second, we want to investigate run-time verification of memory safety. Third, more experimental evaluation can be done on the capabilities and performance of Verified JavaBIP. Fourth and finally, we want to investigate run-time verification of safety properties of the JavaBIP model beyond invariants.

References

1. Abdellatif, T., Brousmiche, K.L.: Formal verification of smart contracts based on users and blockchain behaviors models. In: 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. IEEE (Feb 2018). https://doi.org/10.1109/NTMS.2018.8328737
2. Ahrendt, W., Chimento, J.M., Pace, G.J., Schneider, G.: Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. Form. Methods Syst. Des. **51**(1), 200–265 (Aug 2017). https://doi.org/10.1007/s10703-017-0274-y
3. Amighi, A., Hurlin, C., Huisman, M., Haack, C.: Permission-based separation logic for multithreaded Java programs. Logical Methods in Computer Science **11**(1) (Feb 2015). https://doi.org/10.2168/LMCS-11(1:2)2015
4. André, P., Attiogbé, C., Mottu, J.M.: Combining techniques to verify service-based components (Sep 2022), https://www.scitepress.org/Link.aspx?doi=10.5220/0006212106450656, [Online; accessed 26. Sep. 2022]
5. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3), 329–366 (2004). https://doi.org/10.1017/S0960129504004153
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4^{th} IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM06). pp. 3–12 (Sep 2006). https://doi.org/10.1109/SEFM.2006.27, invited talk
7. Bliudze, S., van den Bos, P., Huisman, M., Rubbens, R., Safina, L.: Artefact of: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java (2023). https://doi.org/10.4121/21763274
8. Bliudze, S., Cimatti, A., Jaber, M., Mover, S., Roveri, M., Saab, W., Wang, Q.: Formal verification of infinite-state BIP models. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis. pp. 326–343. Springer International Publishing, Cham (2015)
9. Bliudze, S., Katsaros, P., Bensalem, S., Wirsing, M.: On methods and tools for rigorous system design. Int. J. Softw. Tools Technol. Transf. **23**(5), 679–684 (2021). https://doi.org/10.1007/s10009-021-00632-0
10. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Exogenous coordination of concurrent software components with JavaBIP. Software: Practice and Experience **47**(11), 1801–1836 (Apr 2017). https://doi.org/10.1002/spe.2495
11. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: IFM. Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017), https://link.springer.com/chapter/10.1007/978-3-319-66845-1_7
12. Bodden, E., Lam, P., Hendren, L.: Partially Evaluating Finite-State Runtime Monitors Ahead of Time. ACM Trans. Program. Lang. Syst. **34**(2), 1–52 (Jun 2012). https://doi.org/10.1145/2220365.2220366
13. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: VeriSolid: Correct-by-design smart contracts for Ethereum. In: Financial Cryptography and Data Security, pp. 446–465. Springer, Cham, Switzerland (Sep 2019). https://doi.org/10.1007/978-3-030-32101-7_27
14. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Electronic Notes in Theoretical Computer Science **55**(2), 255–276 (2001). https://doi.org/https://doi.org/10.

1016/S1571-0661(04)00256-7, RV'2001, Runtime Verification (in connection with CAV '01)

15. Solidity team: Solidity programming language, https://soliditylang.org/, (Accessed at: 2022-10-21)
16. Stulova, N., Morales, J.F., Hermenegildo, M.V.: Reducing the overhead of assertion run-time checks via static analysis. In: PPDP '16, pp. 90–103. Association for Computing Machinery (Sep 2016). https://doi.org/10.1145/2967973.2968597
17. VerifyThis collaborative long-term verification challenge: The Casino example, https://verifythis.github.io/casino/, (Accessed at: 2022-10-12)