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

Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, Larisa Safina

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

Simon Bliudze[1], Petra van den Bos[2], Marieke Huisman[2], Robert Rubbens[2], and Larisa Safina[1]

[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 machine, 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 [8]. 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 [9] 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 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, functions of a JavaBIP program are annotated as FSM transitions. These annotated functions model the actions of the program components. The computations are assumed to be terminating and non-blocking. Furthermore, any side-effect is assumed to be either taken into account 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 the JavaBIP approach is that it does not guarantee that these assumptions hold. This paper proposes a joint extension of JavaBIP and VerCors [10] providing such guarantees about the implementation statically and at run time.

VerCors [10] is a state-of-the-art deductive program verification tool for concurrent programs. It verifies programs, written in e.g. Java, using permission-based separation logic [3]. This logic is an extension of Hoare logic that allows one to specify 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 to include verification annotations, and (ii) adapt VerCors to support JavaBIP annotations. VerCors transforms the verification annotations and JavaBIP models into contract annotations, leveraging their structure as specified by the FSM annotations and the glue.

For some programs VerCors requires extra contract annotations, e.g. when complicated loops are involved. 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. 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 in Verified JavaBIP to prove assumptions made on the JavaBIP models. Specifically, we make the following contributions:

- We extend regular JavaBIP annotations with verification annotations on pre- and postconditions for transitions, and invariants for components and states.
- 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.

2 Related Work

There are several approaches to verify and validate 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 [7]. Abdellatif et al. used the BIP framework to formalize and verify Ethereum smart

```
@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

contracts written in Solidity [2]. Mavridou et al. have introduced the VeriSolid framework, which additionally allows the generation of Solidity code from the verified models [11]. André et al. describe a combined workflow to verify and validate Kmelia models [4]. They also describe the COSTOTest tool, which can run a test harness that interacts with the abstract model. Thus, most of these approaches do not consider verification of model implementation as a primary goal, which is what we aim at with "Verified JavaBIP". Only the COSTOTest tool establishes a connection between the abstract model and the implementation, but it cannot provide any guarantees about memory safety or functional correctness.

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 [9]. Listing 1 shows an example of JavaBIP annotations. The class annotation `@ComponentType` indicates that this class is a JavaBIP component and specifies its initial state. In the example, this is the `WAITING` state. `@Port` declares the list of possible transition labels. Method annotations include `@Transition`, `@Guard` and `@Data`. An `@Transition` annotation consists of the name of a port, a start and end state, and optionally a guard. The transition in the example goes from `WAITING` to `PINGED` whenever the `PING` port is triggered. As no guard is specified, the transition may always be taken. `@Guard` declares a named boolean function, whose result indicates if a transition is enabled. `@Data` either declares outgoing data of a getter method, or, incoming data via a method parameter. Note that the example does not specify when ports are activated. This is specified as glue, separately from the JavaBIP component [9].

We added component invariants and state predicates to Verified JavaBIP. These are declared using the following annotations: `@Invariant(expr)` for a property that must hold after construction of the component and before and after executing a transition, and `@StatePredicate(state, expr)` for property that must hold when entering the state, and is assumed to hold when departing from the state. Furthermore, Verified JavaBIP includes pre- and postconditions as part of the `@Transition` annotation. These have to hold before and after execution of the corresponding function, respectively. Lastly, `@Pure` indicates that a method is side-effect-free, and is used only in conjunction with `@Guard`.

Expressions within annotations should follow the grammar of Java expressions with some limitations: we do not support lambda expressions and method

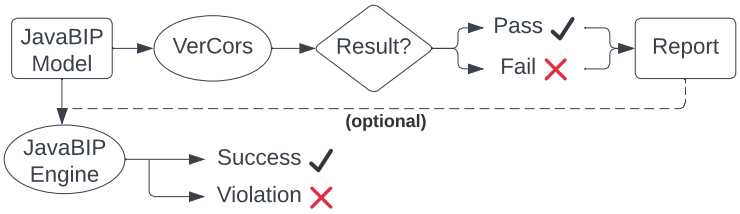

Fig. 1. Verified JavaBIP architecture. Ellipse boxes represent analysis or execution.

references, switch expressions, expressions containing `new` or `instanceOf`, and wildcard arguments. These are limitations of the current implementation. In the future they might be lifted.

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 automatically transform a JavaBIP model to the internal representation of VerCors, Common Object Language (COL), which is roughly equal to the union of the input languages of VerCors. This representation is then verified. An example of this transformation is given in Listings 2 and 3. If verification succeeds, the JavaBIP model is memory safe, has no data races, and all its components respect the specified properties. In this case no extra run-time verification is needed.

If verification fails, there are either memory safety issues, or components violate properties. In the first case, the user needs to update the verification annotations, and retry verification with VerCors, since memory safety properties cannot be checked by the JavaBIP engine. In the second case, VerCors outputs and stores a verification report with the JavaBIP model, indicating which properties were verified and which were not.

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.

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 the relevant properties at points of interest. For example, before the execution of a transition, the pre-condition of the transition as well as the component invariant is checked. Furthermore, the user can configure what should happen if run-time verification fails: print a warning and continue execution, or terminate execution.

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

Listing 2. Example of a transition in JavaBIP.

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

Listing 3. COL representation of Listing 2 after encoding JavaBIP semantics.

Deductive verification is done by encoding the JavaBIP semantics into COL. We describe this encoding with an example. Listing 2 shows the `ping` method, where the `@Transition` annotation indicates a transition from state `PING` to `PING`. The guard indicates that the transition may only be taken if it is not the last ping. The name `HAS_PING` refers to the `@Guard` annotation of the corresponding method `hasPing`, which returns `pingsLeft >= 1`.

Listing 3 shows the COL representation of the `ping` method after encoding the JavaBIP semantics. Line 1 states the precondition of the `ping` method, and line 2 the postcondition. `PING_state_predicate()` refers to the state predicate of state `PING`. It may restrict the values of the fields in the class. If not defined by the user, it is `true`. Since the predicate is both a pre- and a postcondition, it is assumed at the start of the method, and needs to be proven to hold at the end of the method. `hasPing()` is the method with the `@Guard` annotation for the `HAS_PING` label. The method is used directly in the COL representation. For each JavaBIP construct, we have implemented such a transformation of JavaBIP to COL. As these transformations are all similar, the rest of them are not discussed.

To prove memory safety of JavaBIP models, we extended VerCors to automatically generate permission annotations. Currently, a naive policy is used, where each component owns the data of all its fields. This works well for JavaBIP models where each component has full ownership over its data, and none of it is shared. If this is not the case, the naive approach will not suffice, and a different approach will be needed, e.g. by VerCors taking into account user written permissions. For more info about permissions, we refer the reader to [3].

6 VerifyThis Casino and Verified JavaBIP

Finally, we illustrate Verified JavaBIP with the Casino case study adapted from [12]. We discuss the case study, the JavaBIP transformation, and its verification.

The Casino consists of three component types: a player, an operator, and the casino. The model supports multiple players and casinos; each casino has only one operator. The players can place bets on a coin flip. The casino pays out twice for a correct guess, and keeps the money otherwise. The operator can choose to add or withdraw money to the pot of the casino. To decide how much money to add or withdraw, the operator maintains locally the balance of the casino pot.

We have added several invariants to this model. The purse of every player, the casino pot and its operator copy, the wallet of the operator, and a 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` verification annotations on the corresponding components and states in the JavaBIP model (see Appendix A).

There are two problems with this model. However, because their root cause, detection and solutions are very similar, we only discuss one of them. Specifically, the problem is that the player can win more than the casino pot contains. This is possible because there are no restrictions on how much the player can bet.

This problem is detected by both deductive and run-time verification. Concretely, VerCors cannot prove that the casino pot is non-negative, which is part of the casino invariant, after the `PLAYER_WIN` transition. The JavaBIP run-time engine can also detect this problem, but it is not guaranteed to. This is because the model contains some non-determinism. For example, if no player ever wins, this problem does not occur, and run-time verification does not detect it.

There are several possible solutions. First, the user can choose to only run the model with run-time verification enabled, such that the execution can be stopped upon violation. Depending on the model, combined with the performance penalty of run-time verification, this might be an acceptable solution. Second, extra JavaBIP guards can be added to the transitions, restricting model behaviour at run time. For example, a guard could be added to `PLACE_BET` that requires `bet <= pot`. However, in the general case, adding guards might introduce deadlocks, and it slows down run-time verification. Third, a solution is to change the model in such a way that this problem cannot occur. For example, the casino could offer choices of how much the player can bet. This solution introduces no extra run-time checks. However, refactoring a model takes effort, and in the general case the behaviour of the model will change. Depending on the situation, the user may opt for each of these three solutions.

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 with annotations intended for verification of functional properties. Verified JavaBIP offers both deductive verification of models using VerCors, and run-time verification of the model using the JavaBIP engine. Any properties not verified deductively can be checked at run-time, while any properties verified deductively are skipped for checking at run time. In the demonstration, we used Verified JavaBIP on the Casino case study, automatically detecting a problem.

There are several directions for future work. Support for automatically checking memory safety could be further extended by also supporting data-sharing between components. We would also like to support run-time verification of memory safety. Additionally we would like to do more experimental evaluation on the capabilities and performance of Verified JavaBIP. Finally, we are planning to investigate run-time verification of safety properties of the JavaBIP model beyond invariants.

References

1. Solidity programming language, https://soliditylang.org/, (Accessed at: 2022-10-21)
2. 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
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., 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)
8. 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, https://doi.org/10.1007/s10009-021-00632-0
9. 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, https://doi.org/10.1002%2Fspe.2495
10. 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
11. 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
12. VerifyThis collaborative long-term verification challenge: The Casino example, https://verifythis.github.io/casino/, (Accessed at: 2022-10-12)

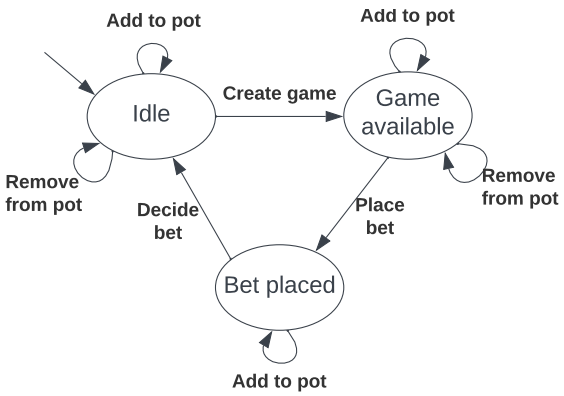

Fig. 2. Finite state machine representation of the Solidity Casino smart contract. Note that this figure does not visualize the structure of the JavaBIP encoding.

Appendix

A Demonstration

This appendix demonstrates the Verified JavaBIP tool set by applying it to the VerifyThis Casino case study. This case study was chosen because it strikes a good balance between being non-trivial, and easy to understand. Additionally, it also contains a problem, detectable with both deductive and run-time verification.

We first discuss the case study and its origin in Appendix A.1, followed by an explanation of how the case study was encoded in JavaBIP, in Appendix A.2. We show how to detect the problem in Appendices A.3 and A.4, and how to fix the problem in Appendix A.5.

Before the artefact deadline, we will provide an artefact containing the following:

- Instructions on the structure and usage of the artefact;
- The source and binary of VerCors with JavaBIP support;
- The JavaBIP Engine with run-time verification support;
- Examples to test the tool set;
- Various versions of the Casino example to illustrate problems with the model and how they can be fixed;
- Scripts to run the tool set on the provided examples; and
- Sources of the tools.

A.1 Solidity Casino: Case Study

The VerifyThis Collaborative Long Term Challenge considers a casino implemented in Solidity, a language for defining smart contracts [1]. This particular implementation of a casino allows players to bet on the outcome of a coin flip. If they guess the correct outcome, they win, and the casino pays out twice the bet. If they guess incorrectly, no money is received, and the casino keeps the money.

Figure 2 visualizes the structure of the casino as a state machine. The operator can add to and remove money from the pot of the casino at any time. Except after a bet is placed, then the operator can only add money to the pot, because placing a bet is only allowed if there is enough money in the pot. If it were possible to remove money from the pot after placing a bet, the casino could end up with a negative balance. For more details, we refer the reader to the VerifyThis website explaining the challenge [12].

A.2 Solidity Casino: JavaBIP Encoding

The JavaBIP encoding of the Casino diverges from the original casino example to generalize the original case study. Concretely, the concept of "operator" and "player" were factored out and made independent. This allows instantiation of the model with any number of casinos, players, and operators. There are several reasons for the generalization. First, a model that can be instantiated for different parameter values makes doing performance evaluations easier. Unfortunately, because of time constraints, this is still future work. Second, the generalized model also allows use of an advanced feature of the JavaBIP framework that was not relevant before, which is that of a three-way synchronization.

We will now discuss the actual JavaBIP encoding. In this encoding, there are three component types: `Player`, `Operator`, and `Casino`. Changes in the state take place as synchronizations between components. For example, when money is added to the pot of the casino, `Operator` and `Casino` synchronize, transferring the amount to be added from the `Operator` to the `Casino`. Similarly, when `Player` bets, `Player` and `Casino` synchronize, communicating the amount to be betted. Finally, when a bet is decided, a ternary synchronization takes place to establish in all three components that the bet was decided. For `Operator`, this means the amount of money in the `Casino` has either increased or decreased. For `Player`, it either receives twice the bet, or nothing. For the `Casino`, it either needs to pay out, or transfer the bet to the pot. The components and their transitions are visualized in Figure 3.

Consider the code in Listing 4, which shows part the `Casino` component. The component starts in state `IDLE`. As indicated by the `@Transition` annotation, the `ADD_TO_POT` transition can be activated in the `BET_PLACED` state. A requirement of the transition is that only the operator can trigger it, which is fulfilled by the guard part of the transition annotation. Besides that, it is always safe to add money to the pot, so it has no pre- and postconditions. In the full JavaBIP model, the method can also be activated in other states, but these annotations have been omitted from the paper for brevity. In the next subsection we will discuss parts of the model that might not be safe, and how these limitations can be expressed in contracts.

The `@Invariant` and `@StatePredicate` annotations in Listing 4 indicate the component invariant and state invariant, respectively. The component invariant indicates that `bet` and `pot` should both contain non-negative integers. The state predicate indicates that in the `IDLE` state, the `bet` variable should be equal to zero. The full JavaBIP model contains more annotations, but they are omitted for brevity.

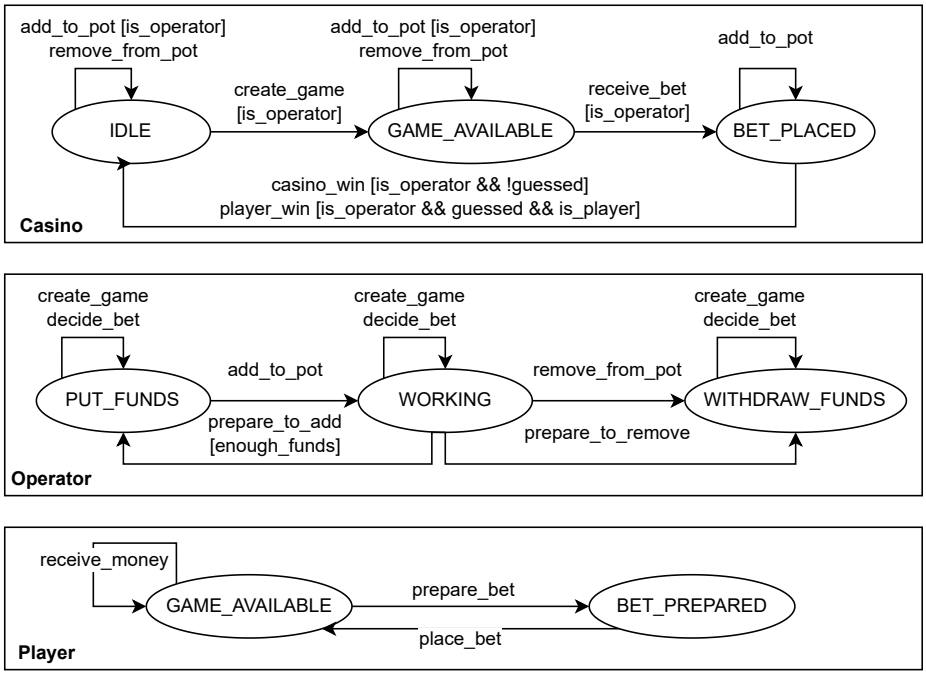

Fig. 3. Finite state machines of components in JavaBIP Casino case study

```
1  @ComponentType(initial = IDLE, name = CASINO_SPEC)
2  @Invariant("bet >= 0 && pot >= bet")
3  @StatePredicate(state = IDLE, expr = "bet == 0")
4  public class Casino {
5    int pot;
6    Coin guess;
7    int bet; // Other variables omitted...
8
9    @Transition(name = ADD_TO_POT, source = BET_PLACED, target
         = BET_PLACED, guard = IS_OPERATOR)
10   public void addToPot(@Data(name = OPERATOR) Integer sender,
          @Data(name = INCOMING_FUNDS) int funds) {
11         pot = pot + funds;
12         System.out.println("CASINO" + id + ": " + funds +
13         " received from operator " + sender +
14         ", pot: " + pot);
15   }
16   // Rest of component...
17 }
```

Listing 4. Excerpt of JavaBIP Casino component

```
====================================
100   // Remove money from pot
101   @Transition(name = REMOVE_FROM_POT, source = IDLE,
    target = IDLE, guard = IS_OPERATOR)
102   public void removeFromPot(@Data(name = OPERATOR)
    Integer sender, @Data(name = INCOMING_FUNDS) int funds) {
103     pot = pot - funds;
104     System.out.println("CASINO" + id + ": " + funds +
105     " removed by operator " + sender +
106     ", pot: " + pot);
107   }
------------------------------------
In this transition the invariant of the component is not
    maintained, since ...
------------------------------------
                                [----------
  2   @Invariant("bet >= 0 && pot >= bet")
                                 ----------]
------------------------------------
... this expression may be false
====================================
```

Listing 5. Output of VerCors after verifying the Casino JavaBIP model.

Even though the model was carefully designed, it contains a problem: the operator can withdraw more money than is available in the `Casino` pot. We will now show how this problem can be detected with both deductive and run-time verification.

A.3 Deductive Verification

To deductively verify the Casino model with VerCors, the source code of the model is given to VerCors:

```
$ vercors Casino.java Constants.java Operator.java Player.
    java Main.java
```

After several seconds of analysis, VerCors reports that the invariant of the `Casino` component does not hold after removing money from the pot, as shown in Listing 5.

Note how both the transition that violates the invariant is shown, as well as which part of the invariant is violated. As `bet` is supposed to be non-negative, and `pot` should be greater or equal to `bet`, it follows that VerCors cannot prove that `pot` remains non-negative.

A.4 Run-time Verification

The JavaBIP engine with run-time verification now checks the contract that, according to the VerCors output, it was not able to verify: `pot >= bet`.

In the following we see that the execution starts with creating the casino, and initializing the operator and players with the predefined amount of money in their wallets. After that players make their first bet.

```
OPERATOR101 created with wallet: 500
CASINO201: INITIALIZED
PLAYER301: INITIALIZED
PLAYER302: INITIALIZED
PLAYER303: INITIALIZED
OPERATOR101: decided to put 446, wallet: 54
PLAYER303: bet 6 prepared, purse: 94
PLAYER301: bet 48 prepared, purse: 52
PLAYER302: bet 20 prepared, purse: 80
CASINO201: GAME CREATED
...
```

After receiving the bets, the Casino is checking the guesses and concludes that player 2 wins! However, this makes the pot to be negative, which triggers an alert from the invariant violation.

```
CASINO201: received bet: 20, guess: TAILS from player 302
CASINO201: 20 lost, pot: -20
19:06:54.881 [ACTOR_SYSTEM-akka.actor.default-dispatcher-3]
ERROR org.javabip.executor.BehaviourImpl - Invariant
    violation: bet >= 0 && pot >= bet
```

A.5 Fixing the problem

As discussed in Section 6, there are several ways to fix a broken model: (i) always execute the model with run-time verification, (ii) add extra guards, or (iii) refactor the model. For the sake of illustrating the tool set, we will show an example of applying option (ii). Concretely, we add the guard `ENOUGH_FUNDS` to the transition annotation of `REMOVE_FROM_POT`. The updated annotated method is shown in Listing 6. This will restrict the transition to only be enabled when there is enough money in the casino pot, ensuring that the casino pot cannot become negative because of withdrawals from the operator.

For deductive verification, the change in the model causes VerCors to succesfully verify the model: it prints "`Verification completed successfully.`"

```
1 // Remove money from pot
2 @Transition(name = REMOVE_FROM_POT, source = IDLE, target =
      IDLE, guard = "IS_OPERATOR && ENOUGH_FUNDS")
3 public void removeFromPot(@Data(name = OPERATOR) Integer
      sender, @Data(name = INCOMING_FUNDS) int funds) {
4   pot = pot - funds;
5   System.out.println("CASINO" + id + ": " + funds +
6     " removed by operator " + sender +
7     ", pot: " + pot);
8 }
```

Listing 6. `REMOVE_FROM_POT` transition with the new guard underlined.

upon termination, which indicates that the model is memory safe, data race free, and respects the verification annotations. Furthermore, a verification report is produced, which states that the invariant of the Casino component has been verified. For run-time verification, the change will cause the model to no longer crash at run-time when run-time verification is enabled, as the invariants are no longer violated. Additionally, when the previously produced verification report is included when executing the model, the run-time verifier skips checking the component invariant of Casino while running the model. For a small model such as this one, the difference between doing the run-time checks and not doing them is negligible. However, we speculate that for models with more components, the overhead could be noticeable.