A Model-Based Approach to Combining Static and Dynamic Verification Techniques*

Shaun Azzopardi, Christian Colombo, and Gordon Pace

University of Malta

Abstract. Given the complementary nature of static and dynamic analysis, there has been much work on identifying means of combining the two. In particular, the use of static analysis as a means of alleviating the overheads induced by dynamic analysis, typically by trying to prove parts of the properties, which would then not need to be verified at runtime. In this paper, we propose a novel framework which combines static with dynamic verification using a model-based approach. The approach allows the support of applications running on untrusted devices whilst using centralised sensitive services whose use is to be tightly regulated. In particular, we discuss how this approach is being adopted in the context of the Open Payments Ecosystem (OPE) — an ecosystem meant to support the development of payment and financial transaction applications with strong compliance verification to enable adoption by payment institutions.

1 Introduction

Analysis of the dynamic behaviour of systems has long been used to ensure the correct behaviour of software. By adding monitors, for instance in the form of assertions, throughout the system, one can react accordingly whenever unexpected behaviour occurs. One issue with this is that of monitoring overheads. Monitors impose a temporal burden on the system, and in the case of properties which go beyond the well-formedness of the system state at a particular point in time (an example of such a property would be 'File access can only occur between a login and a logout'), such checks also require additional space overheads to store the state of the monitor. One prevalent approach in addressing these overheads has been that of applying static analysis techniques, typically in an attempt to prove parts of the specification which would not need to be checked at runtime.

Dynamic verification also fails to address certain verification scenarios. Consider a situation in which a service would need to be matched with an appropriate co-service. For example, an online sales application providing its services via an e-commerce portal may need to be paired with a postal service provider and a payment institution to perform payment transactions . However, some postal service providers may only provide their services to e-commerce applications

* The Open Payments Ecosystem has received funding from the European Union's Horizon 2020 research and innovation programme under grant number 666363.

whose customers are from a particular geographic region, and certain payment institutions may only allow applications which store certain information about customers and have a guaranteed minimum payment throughput. Since the allocation of postal and payment service providers has to take place before the application is deployed, dynamic verification provides such information too late. In such a situation, static analysis can be used to address the issue of matching appropriate service providers with the application. However, one would typically expect such applications to run on the user's machine, making it hard to ensure that the analysed application is the one actually deployed.

Model-based approaches for the analysis of computer systems have long been adopted in various settings. In these settings, the notion of what constitutes a model is rather fluid, but is typically taken to be a description of the system such that: (i) it behaves in a manner which is faithful (with respect to certain features) to the system itself; but (ii) it abstracts away other unnecessary detail, hence making it more amenable to analysis. Typically, whether it is used for simulation, testing or model checking, the approach allows the analysis of the model prior to deployment. The underlying degree of trust that the model is similar to the actual system means that conclusions can be carried over from the model to the actual system. However, models do not always carry the same degree of trust — while a model deduced from a system using a verified-correct algorithm (e.g., extracting the control-flow graph of a system) can be guaranteed to be correct with respect to the information retained, other models (e.g., a UML model based on which a system is developed in an *ad hoc* manner) may not be faithful to the actual system.

In this paper, we present a model-based framework for the verification of systems which require analysis before deployment. An example of such a scenario is when one is required to match a system with compatible co-services (henceforth referred to as *service providers*) based on a model of the system as given by the developer (and thus not verified). To ensure behavioural correctness one thus also needs to runtime verify compliance of the actual application behaviour against the model.

This need for such an approach arose in the context of a platform for the deployment of financial systems which require services provided by financial institutions such as banks [3]. This brings in various constraints from the service providers: (i) capabilities e.g., which credit cards they can handle; (ii) legislation in countries they operate in e.g., anti-money laundering legislation places limits on individual spending depending on how much knowledge about the customer the financial institution has acquired (usually referred to as *customer due diligence*); and (iii) what risks they are willing to take (*risk appetite*). Applications submitted by registered developers to the payments platform would thus need to be analysed *prior to deployment* to ensure they are paired up with an appropriate service provider and also checked for regulatory compliance. However, an additional challenge is that developers are not constrained to a particular technology, and applications access the financial platform through a generic API, meaning that analysis and verification techniques cannot be technology-specific.

The solution adopted is to have developers submit a model of their application's use of API calls which (i) is analysed to allow pairing up the application with a service provider; (ii) is used to (usually partially) verify the application against regulatory compliance. However, since the implementation cannot be trusted, the application behaviour is verified at runtime to ensure that it adheres to the model the developer submitted.

The solution we present in this paper uses a model-based approach to achieve various goals. Firstly, we ensure technology independence — by using the model for static verification instead of the actual system itself. Secondly, we also use the model to attempt to verify compliance properties statically, and although the model is typically too weak to ensure full compliance, it can serve to prove parts of them, reducing overheads induced due to runtime verification of these properties. Finally, since the system might not be a correct refinement of the model, we also verify the model at runtime.

The paper is organised as follows. We start by presenting static and dynamic analysis approaches in Section 2. Next, Section 3 provides an overview of our proposed solution, while Section 4 delves into an instantiation of our framework as a case study. Finally, we frame our work in the context of the OPE project in Section 5, and conclude in the last section.

2 Combining Static and Dynamic Verification Techniques

Much of the literature combining static and dynamic verification decomposes the specification π of a system P in such a way that part of the specification is statically verified at compile time, leaving the rest to be verified at runtime. At a most basic level, one takes conjuncts and sees which of these can be verified statically e.g., [2, 4], leaving the rest to be verified at runtime:

$$\dfrac{\mathrm{SA}(P,\pi_1)\ \dfrac{}{P \vdash \pi_1} \quad \mathrm{RV}\ \dfrac{}{P \vdash \pi_2}}{P \vdash \pi_1 \wedge \pi_2}$$

In the pseudo-rule above, we abuse notation and use a family of static-analysis pseudo-axioms SA(P, π) which asserts that by using a static analysis technique to check whether program P satisfies property π, we manage to automatically prove the entailment of the proof rule[1]. Similarly, we use the pseudo-axiom RV to indicate that the entailment will be verified at runtime.

The approach is thus to separate the conjuncts of the property which are verifiable statically (property π_1) from those which are not, and thus have to be verified at runtime (property π_2). In most cases, monitoring the structurally smaller formula π_2 induces less overheads than $\pi_1 \wedge \pi_2$, which is desirable. Note that there are cases where this is not necessarily true. For instance, monitoring

[1] We show P and π as parameters to specify exactly what the verification question posed to the static analysis tool is.

a predicate $P \implies Q$ might be more costly than monitoring $(P \implies Q) \wedge \neg P$ which a pre-processor might simplify to $\neg P$.

In fact, what is required to be monitored can be weakened to take into account that there may lie an overlap between the conjuncts. As we will show, if we know that π_1 is satisfied by the system, and we want to ensure that $\pi_1 \wedge \pi_2$ holds, it suffices to monitor any predicate α which satisfies (i) $\pi_1 \wedge \pi_2 \implies \alpha$; and (ii) $\alpha \implies \neg\pi_1 \vee \pi_2$. Trivial solutions for α include $\pi_1 \wedge \pi_2$ and π_2, but other solutions might be better suited for dynamic verification.

This view of the combination of static and dynamic analysis can be somewhat generalised so as not to be limited to static analysis, which can prove a sub-conjunct of the specification. In this case, if when trying to verify a specification π, the static verifier manages to prove property π', the runtime verification tool will have to verify what remains of the original specification modulo what was proved. This can be expressed using the notion of property quotients.

Definition 1. *Given predicates α and β, a predicate γ is said to be a* quotient of α with respect to β, *written as $\gamma \in \alpha \div \beta$, if $\beta \implies (\alpha \iff \gamma)$.*

Quotients are defined in such a way that if $\gamma \in \alpha \div \beta$, then to prove α when knowing β, it suffices to prove γ. Furthermore, knowing γ is necessary for α to hold (under β). For instance, if $\gamma \in P \wedge Q \div R \vee Q$ would mean that γ satisfies $(R \vee Q) \implies ((P \wedge Q) \iff \gamma)$ — with $P \wedge Q$ and $P \wedge (R \implies Q)$ being two possible solutions for γ.

The following proposition gives us two inequalities (implications) specifying the possible values, which the quotient of a specification with respect to a known property can take.

Proposition 1. *Given a specification S and known property π, if $\pi' \in S \div \pi$, then it follows that: (i) $S \wedge \pi \implies \pi'$; and (ii) $\pi' \implies S \vee \neg\pi$. Furthermore, for these implications to hold, it is necessary that π' is a quotient of S with respect to π.*

Proof. *Since π' is a quotient of S with respect to π, we know that: $\pi \implies (S \iff \pi')$. From this, it is straightforward to prove that $S \wedge \pi \implies \pi'$. The second implication $\pi' \implies S \vee \neg\pi$ also holds by case analysis on π'.*

For the second part of the proof, assume that these two implications hold. If π holds, from the first we obtain that $S \implies \pi'$, and from the second we get $\pi' \implies S$, from which we conclude that $S \iff \pi'$.

We can use quotients to extend the verification rule to take into account whatever the static verification analysis managed to prove. When checking for a specification S, if a property π_1 is known (can be proved statically), we can dynamically check any property which is (under the effect of π_1) equivalent to S — in other words, any quotient of S with respect to π_1:

$$\dfrac{\mathrm{SA}(P,\pi)\ \dfrac{}{P \vdash \pi_1} \quad \mathrm{RV}\ \dfrac{}{P \vdash \pi_2}}{P \vdash \pi}\ \pi_2 \in \pi \div \pi_1$$

Note that throughout this section, we have not committed ourselves to any particular specification logic, as long as it has conjunction and negation operators, and forms a Boolean algebra.

2.1 Related Work

StaRVOOrs [2, 1, 5] is a tool that reflects most closely the framework presented. In this framework, the static analysis tool KeY attempts to prove parts of specification by theorem proving pre-/post-conditions. The result of this analysis is to either (i) completely discharge conjuncts of the specification (similar to the initial approach we described in this section); or (ii) discharge parts of the specification along particular branches of execution (which corresponds to a quotient as defined above).

A residual specification is not always explicit, for example in the Clara framework [4], static analysis for finite-state properties are used to disable AspectJ monitors at program locations that have been proven to be safe. This is thus aimed at leveraging several static analyses to reduce monitoring overheads.

[8] illustrates the logic behind such an approach, wherein static analysis is leveraged to specify regions of a program that are safe, i.e., within which instrumentation can be avoided. If a certain method call is always present in such a safe region, then it is also removed from the specification, leaving a residual specification to be instrumented within the potentially unsafe regions of the program.

Another approach allows avoiding monitoring overheads completely by simply replacing these execution points by halt statements instead of weaving monitors at potentially unsafe locations [12]. This allows for the creation of a modified program that is safe, and that has no added overheads during runtime, although some (potentially unsafe) execution traces will simply halt the execution.

3 A Model-Based Approach

Under certain circumstances, it may be impractical if not impossible to have access to the actual system to analyse statically. For instance, consider an application accessing a sensitive central resource or service, which would itself be deployed on an untrusted device. Since there is no guarantee that the code executed is not tampered with, analysing the given code would be a futile exercise. Another consideration is that one might want to allow an application to be implemented using any of a wide range of technologies, accessing the sensitive resource through an API. Developing static analysis for any source technology would be prohibitively expensive and impractical. In both these examples, it is interesting to note that the use of the sensitive resource through a centralised server ensures that, provided that the properties are limited to the actual resource usage, dynamic analysis is still feasible, even if pre-deployment static analysis is not, thus precluding the use of techniques such as the ones described in the previous section.

One way of addressing this challenge is the use of models. By having access to a model of the system — an abstract description of how the system will behave with respect to the central resource — one can perform static analysis of the properties using the model rather than the system itself. However, if the property is verified statically against the model, we would still have no guarantee that the system actually behaves as promised by the model, a guarantee which for the reasons expounded earlier, we can only check dynamically at runtime.

We can extend the approaches presented in the previous section to enable the use of a model M (of which system P is supposed to be a refinement, written as $P \sqsubseteq M$). If the model is sufficient to prove property π, at runtime it suffices to verify that the system is a refinement of the model:

$$\frac{\text{SA}(M,\pi)\ \dfrac{}{M \vdash \pi} \quad \text{RV}\ \dfrac{}{P \sqsubseteq M}}{P \vdash \pi}$$

In this approach, the model acts as both a description of the system (during the static analysis) and the property (for the dynamic analysis).

This approach poses a number of challenges. The first challenge is the obvious question of why we should put any weight on the promise of the model. After all, nothing stops the developer from submitting a 'perfect' model and then deploying an application which behaves in a completely different manner. Although we would realise the misbehaviour at runtime, and stop the application from proceeding once it diverges from what the model promised, what use was the static analysis? Wouldn't the alternative approach of just monitoring the property have been equally effective? In practice, however, the justification of the use of the model is twofold:

(i) In some scenarios, we would like to perform some pre-deployment static analysis to match the application with an appropriate service provider. Consider matching an application performing financial transactions, which would need to be matched with a service provider able to handle transactions of the size and volume that the application will produce, from the geographical locations supported by the application, providing the right financial instruments required by the application, etc. Unless we have a model of the application based on which an appropriate service provider can be found, we would not be able to proceed any further. Moreover, the fact that some service providers are not willing to engage with an application which does not promise to behave correctly, ensures that the developer is willing to provide such a model.

(ii) If an application fails to behave according to the model, appropriate action can be immediately taken, stopping or rectifying the problem. However, such misbehaviour can also be tagged; potentially leading, for instance, to a service provider refusing to supply a service to applications by the same developer in the future. This encourages developers to ensure that their applications conform to submitted models.

Providing models of applications, would thus be feasible in a service-oriented ecosystem. Another challenge is that the model abstracts detail without which certain properties are impossible to verify completely. In such situations, we would have to resort to techniques such as those described in the previous section to runtime verify residual parts of the properties. Whereas before, parts of the properties might have been unprovable due to limitations of the static analysis techniques, in this case, unprovable properties may be the result of a weak model.

Using a more formal syntax, given a specification π which is not completely provable from model M, we will use the quotient of π with respect to the property π_1 which the static analyser manages to prove:

$$\frac{\mathrm{SA}(M,\pi)\ \frac{}{M \vdash \pi_1} \quad \mathrm{RV}\ \frac{}{P \sqsubseteq M} \quad \mathrm{RV}\ \frac{}{P \vdash \pi_2}}{P \vdash \pi}\ \pi_2 \in \pi \div \pi_1$$

The architecture of this approach, when the system is potentially executed on an external address space is shown in Fig. 1(a), in which the system is runtime verified against the model and the residual specification by instrumenting the server to arbitrate the system's interaction with the central system.

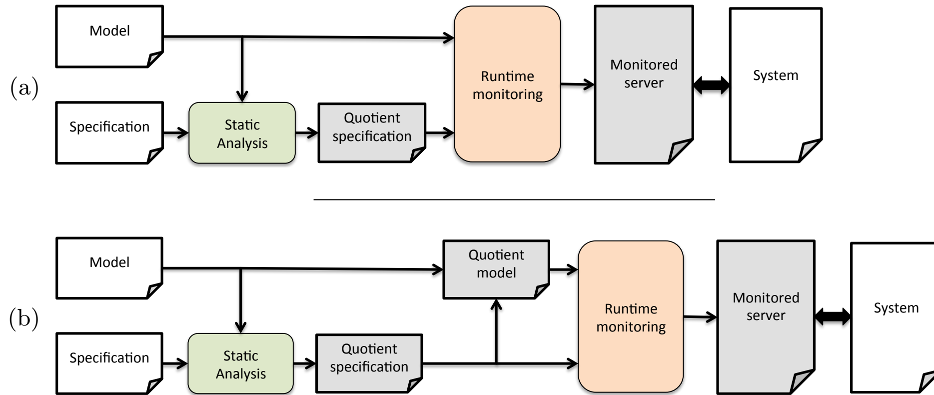

Fig. 1. (a) Architectures for model-based hybrid verification (top); (b) Extended architecture (bottom)

Furthermore, since we are checking that the quotient (π_2) holds, parts of the model M might be discarded since they are being verified by checking for π_2. If the model has a semantics which is comparable to the property language (e.g., for a model M and property π, the formulae $M \wedge \pi$ and $M \implies \pi$ can be computed in the form of a model), we can express this in terms of the quotient operator:

$$\frac{\text{SA}(M,\pi)\ \overline{M \vdash \pi_1} \quad \text{RV}\ \overline{P \sqsubseteq M'} \quad \text{RV}\ \overline{P \vdash \pi_2}}{P \vdash \pi}\ \pi_2 \in \pi \div \pi_1,\ M' \in M \div \pi_2$$

Such approaches have already been explored in other contexts such as [7].

The architecture of the resulting framework is shown in Fig. 1(b).

An example of this in action is [11], where model checking technology is extended to allow the partitioning of a model into two parts, a part free of errors and the rest. This allows for analyses to focus on the second part, further refining it. In effect this can be seen as an implementation of a model quotient with respect to the parts of the specification that the static analysis is able to prove.

4 A Control-Flow-Based Use Case

We will now look at instances of the model-based approach described in the previous section. In this section, we will be using automaton-based formalisms for both the model and the property. We will model the system using a control-flow graph of an application, while we will use a formalism based on the specification language used by the runtime verification tool Larva to write the properties. Despite the fact that both formalisms are given in terms of an automaton, it is important to note that they have different semantics. Let us start by presenting the formalism used to specify the model:

Definition 2. *A simple control-flow model M is a quadruple $\langle \Sigma, Q, Q_0, \rightarrow \rangle$, with initial states $Q_0 \subseteq Q$ and transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$, and is interpreted as a labelled transition system. We will write the reflexive transitive closure of $\rightarrow$ as $\overset{w}{\Rightarrow}$ (with $w \in \Sigma^*$). The set of traces accepted by M, written $traces_M(M)$ is defined to be $\{w : \Sigma^* \mid \exists q_0 \in Q_0,\ q \in Q\ \cdot\ q_0 \overset{w}{\Rightarrow} q\}$.*

Model refinement is defined in terms of trace semantics: $M \sqsubseteq M' \overset{df}{=} traces_M(M) \subseteq traces_M(M')$.

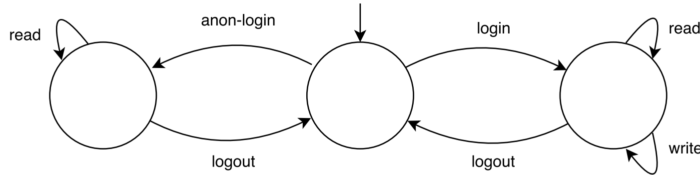

Fig. 2. Model M of the system showing no writes will be allowed during an anonymous login.

For instance, the simple control-flow model, referred to in the rest of this section as M given in Fig. 2, models the control-flow behaviour of the system e.g. that the system will not write just after an anonymous login occurs, and will not perform two consecutive logouts, etc. We will now discuss how various properties can be checked for a system modelled by M.

Our properties in this section will be expressed using a simplified form of dynamic automata with timers and events (DATEs) as used in the runtime verification tool Larva [6]. These automata (which can be dynamically replicated) will have a number of bad states (marked by a cross). Transitions are labelled by triples: (i) a system event which triggers the transition; (ii) a condition which is checked whenever the event is triggered — if the condition holds, then the transition is enabled, otherwise it is not; and (iii) an action which is executed just before the transition is taken. The semantics of such properties are taken to be the set of traces which do not lead to a bad (crossed) state.

The full semantics of DATEs can be found in [6], but for the sake of this example, we will use a semantics over traces of pairs of events and system state snapshots (the latter, accessed through the use of aspect-oriented programming techniques, are required in order to be able to reason about conditions and actions which may refer to the state of the system), with the trace semantics of a DATE property π being written as $\text{traces}_D(\pi)$.

To avoid busy diagrams, we implicitly include reflexive transitions such that if an action happens while in a state in which the action does not appear on any outgoing transition from that state, the automaton allows it and remains in the same state. Consider property π_1 which states that *'a login may only happen while logged out'* which is described using the DATE in Fig. 3(a).

A straightforward way of statically model checking whether a simple-control flow model satisfies a DATE property is by first (i) abstracting the DATE by discarding conditions and actions, thus ending up with an over-approximation of the property, and then (ii) taking the synchronous product of the property automaton and the model (M) to statically verify that no paths in the language of traces generated by the model lead to a bad state. This static analysis approach algorithm can be shown to be sound, though obviously not complete.

This static analysis suffices to show that π_1 is satisfied by M, which will allow us to discard completely the monitoring of the property.

Let us consider a more complex property π_2 which limits the total amount of data transferred while anonymously logged in, as shown in Fig. 3(b). Clearly, the model is not sufficient to guarantee the property. However, the model guarantees that no writing will ever take place while anonymously logged in.

Consider the restriction of a DATE property π with respect to a simple-control flow model M which works by discarding infeasible transitions through the semi-synchronous composition of π with M, akin to the approach used in [10, 9]. This can be shown to yield a quotient of M with respect to π.

We can thus use this quotient of the property with respect to what can be proved based on the model. One solution allowed by our framework is to discard

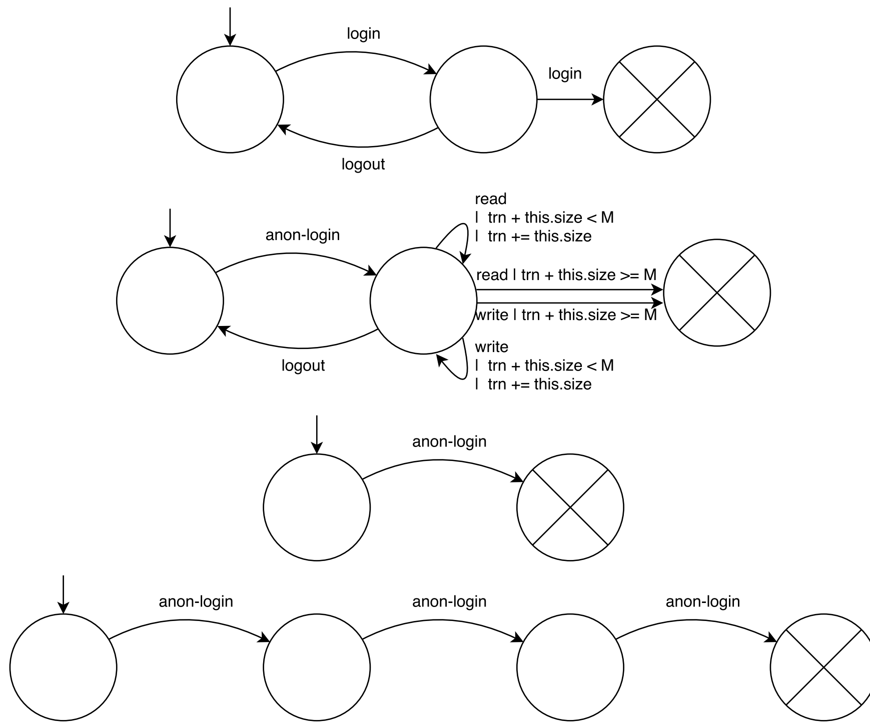

Fig. 3. (a) π_1 = A login may not happen when already logged in; (b) π_2 = No more than X bytes may be transferred while anonymously logged in; (c) π_3 = Only logins with full credentials are allowed; (d) π_4 = The system should allow no more than 2 anonymous logins.

the property transitions tagged by *write* as shown in Fig. 4(a), an optimisation which can have a substantial impact in reducing monitoring overheads.

Finally, consider the property π_3 which states that *'Anonymous logins are not to be allowed'* as shown in Fig. 3(c). Based on the model, which does not rule out anonymous logins, the property cannot be simplified using the static analysis approach shown earlier. However, given that the DATE contains no conditions or actions, it is possible to apply the restriction algorithm mentioned earlier to restrict M to π_3 (which will anyhow be dynamically verified). The resulting model, shown in Fig. 4(b), ignores anonymous logins since these would be detected as violations of the property anyway. At runtime, it still remains to be verified that the system P is really modelled by M reduced with respect to π_3: $P \sqsubseteq M'$, where $M' \in M \div \pi_3$.

As a final example, consider property π_4 stating that "The system should allow no more than 2 anonymous logins", shown in Fig. 3(d). As in the case of π_3, no part of the property can be verified with respect to the model. However, when computing the quotient model as done before, we can calculate a number

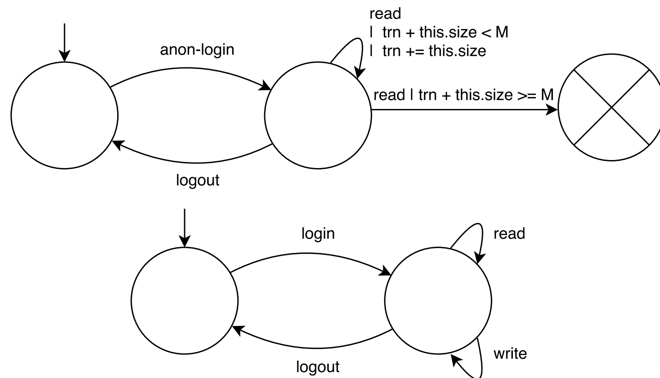

Fig. 4. (a) Property π_2 simplified with respect to model M; (b) Model M simplified with respect to property π_3.

of possible quotients — from the trivial solution of keeping M to expanding the model to keep count of transitions, obtaining a model as shown in Fig. 5. Although the model is larger, it can permit switching off monitors for anonymous logins after the first two of such logins. Depending on the way monitoring is implemented, this might be beneficial.

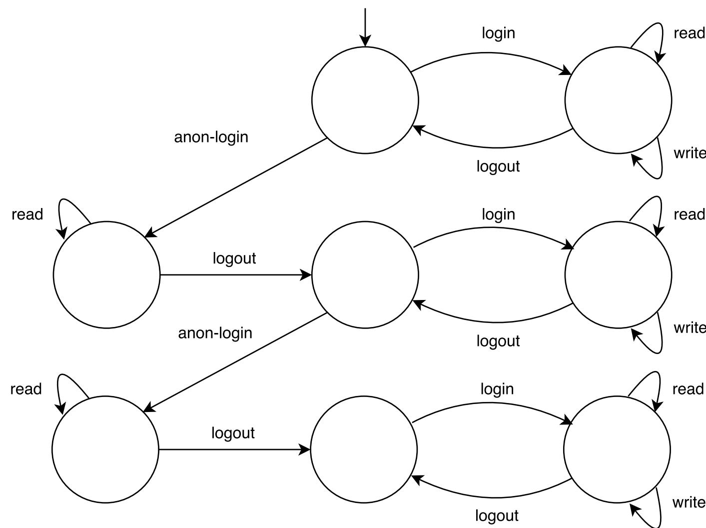

Fig. 5. Model M simplified with respect to property π_4.

5 Open Payments Ecosystem: A Real-Life Case Study

Businesses often find themselves needing diverse ways of affecting or enabling payments in various contexts. As an example, consider a business providing a payment service to a travel agency to purchase flights, hotel bookings, etc. Having several such purchases from a single corporate card, particularly if that same card is also used for other purchases, would make reconciliation non-straightforward at best. On the other hand, providing one shot cards for use by the travel agency, which are cards that can be used once and disabled after the first purchase, makes reconciliation easier as only one purchase will be associated with any given card. However, for a business to set up such a payment programme is quite complex (implement cards processes for provisioning, reconciliation, dispute management, as well as creating a compliant application) and the costs may be prohibitive.

Open Payments Ecosystem (OPE) aims at building an infrastructure to address this need by building an *execution environment* for financial transactions where the process of deploying custom payment applications is simplified. OPE, thus brings together a number of players: (i) *developers* who create the payment applications; (ii) *service providers* (typically banks) which affect the underlying financial transactions; (iii) *corporate customers* (the travel agent in our example) who in turn provide the payment applications to their customers; and (iv) *program managers* who take responsibility of putting programs together — combining applications to service providers — and provide them to corporate customers.

In order to support developers, OPE provides a *development environment* with the necessary APIs for application development and service provider integration. The OPE itself does not hold funds (which legally, can only be held by a regulated institution). Therefore, applications developed are submitted to the OPE, and can be adopted by programme managers who rely on an automated *compliance check* to pair the adopted application with an integrated service provider to enable its execution on the OPE platform. We note that ultimately, service providers carry the financial and regulatory liability for the services offered, and therefore having a reliable automated compliance process goes a long way in enabling service providers to operate with more peace of mind.

The compliance subsystem at the core of OPE has multiple roles: (i) it is used to support programme managers when matchmaking an application and a service provider; (ii) it ensures that a programme does not violate national legislation and regulations based on the location where it is planned to be deployed; and (iii) provides runtime monitoring on the running programme to continually check whether the monitored constraints are violated.

Since the OPE architecture envisages that the payment application is executed outside the platform (typically on the end user's device or a web server), only accessing the OPE platform through API calls, it is possible that the application submitted for validation and matched with an appropriate service provider is compromised or tampered with. Furthermore, providing compliance

algorithms which support different programming languages and technologies, which developers may adopt, is not scalable in the long run.

In this context, the approach presented earlier in the paper becomes useful: developers are expected to submit a sufficiently detailed model of the application behaviour, enabling matching with the service providers and preliminary checking of compliance to applicable legislation. The *Payment Application Modelling Language* (PAML) is a domain-specific language developed specifically to enable the description of a model of a payments application — its components, their attributes, and how the components can interact together.

Example 1 (Model). As a running example, consider a simple model of an application allowing customers to have managed cards, with a total maximum transactional value not exceeding £2,000 per month, with the possibility of depositing and redeeming[2] of funds.

This model is then subjected to the first round of checking, i.e., static analysis, of two types of constraints: that it adheres to the applicable legislation and that it is acceptable to the service provider who will run the application.

Example 2 (Static analysis). In the example model, according to legislation, the customer should always have the possibility of redeeming the money remaining in his or her account after closure. Using the model, we can statically check that this possibility is in fact supported. At this point, we note that regulations also state that money redemption should occur *at par value* and *without delay.* However, it is not possible to statically verify that these hold as the model does not contain this level of detail.

Once the application goes through the static analysis phase and is deployed, we runtime verify that the implementation is indeed a refinement of the model, and that it adheres to the regulations which could not be checked statically.

Example 3 (Runtime verification). Ensuring that the model is a true representation of the implementation in this example entails checking that the application does not exceed the £2,000 monthly limit on the user's transactions.

From the legal side, as noted earlier, we have to ensure that redemption takes place at par value, i.e., the customer receives the right amount, and without delay. These checks are both delegated to be carried out through runtime monitoring.

These are the types of checks we have mostly encountered in the context of OPE. We are currently looking into ways of enriching static analysis to be able to check parts of properties which involve summing and counting values. While the analysis techniques presented in the running example sound simple, we believe that their extension can be effective in lowering the overheads in dynamic verification, while still ensuring safe matching of applications with service providers.

[2] Redeeming funds refers to the withdrawal of electronic money by the customer following the closure of his or her account.

6 Conclusions

The synergy afforded by the combination of static and dynamic analysis has been well studied under the assumption that the implementation is available during the static phase. As motivated by the OPE case study, this is not always the case, particularly when the code is run by an untrusted third party. Even when the analysis of the implementation is feasible, it might not be desirable to support the analysis of a plethora of different technologies.

By statically checking a model of the implementation rather than the implementation itself, we bypass the issues of untrusted and technology-varied code, requiring, however, the runtime checking of the adherence of the implementation to the model. Using the interplay of static and dynamic analysis once more to our advantage, we have shown how the checking of the model can also be simplified by exploiting the residual checks which will anyway be checked at runtime.

To illustrate the proposed approach, we have provided an instantiation based on control-flow models and properties, as well as how the theory is being used in practise in the context of the OPE.

The presented work leaves a number of venues for future exploration, not least how to calculate an efficiently monitorable residual both for checking the compliance to the model and the checks which could not be statically verified on the model. We also plan to implement the complete framework as part of the OPE project, possibly integrating existing tools in the process.

References

1. W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. A specification language for static and runtime verification of data and control properties. In *FM'15*, volume 9109, pages 108–125. 2015.
2. W. Ahrendt, G. Pace, and G. Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISOLA'12*, LNCS 7609. 2012.
3. S. Azzopardi, C. Colombo, G. J. Pace, and B. Vella. Open payments ecosystem. In *Computer Science Annual Workshop 2015 (CSAW'15)*. University of Malta, Nov. 2015.
4. E. Bodden, P. Lam, and L. J. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *RV'10*, volume 6418 of *LNCS*, pages 183–197, 2010.
5. J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS: A tool for combined static and runtime verification of java. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 297–305, 2015.
6. C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, pages 135–149, 2008.
7. F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. *Fundam. Inform.*, 51(4):339–368, 2002.

8. M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 124–133, New York, NY, USA, 2007. ACM.
9. S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, pages 186–196, 1990.
10. J. Krimm and L. Mounier. Compositional state space generation from LOTOS programs. In *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, pages 239–258, 1997.
11. A. Lal, N. Kidd, T. Reps, and T. Touili. Abstract error projection. In H. R. Nielson and G. Filé, editors, *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 200–217, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
12. D. Wonisch, A. Schremmer, and H. Wehrheim. Zero Overhead Runtime Monitoring. In *SEFM'13*, volume 8137 of *LNCS*, pages 244–258. Springer Berlin Heidelberg, 2013.