A Specification Language for Static and Runtime Verification of Data and Control Properties*

Wolfgang Ahrendt[1], Jesús Mauricio Chimento[1], Gordon J. Pace[2], and Gerardo Schneider[3]

[1] Chalmers University of Technology, Sweden.
ahrendt@chalmers.se, chimento@chalmers.se
[2] University of Malta, Malta.
gordon.pace@um.edu.mt
[3] University of Gothenburg, Sweden.
gerardo@cse.gu.se

Abstract. Static verification techniques can verify properties across all executions of a program, but powerful judgements are hard to achieve automatically. In contrast, runtime verification enjoys full automation, but cannot judge future and alternative runs. In this paper we present a novel approach in which data-centric and control-oriented properties may be stated in a single formalism, amenable to both static and dynamic verification techniques. We develop and formalise a specification notation, *ppDATE*, extending the control-flow property language used in the runtime verification tool Larva with pre/post-conditions and show how specifications written in this notation can be analysed both using the deductive theorem prover KeY and the runtime verification tool Larva. Verification is performed in two steps: KeY first partially proves the data-oriented part of the specification, simplifying the specification which is then passed on to Larva to check at runtime for the remaining parts of the specification including the control-centric aspects. We apply the approach to Mondex, an electronic purse application.

1 Introduction

Runtime verification and static verification are widely used verification techniques. *Runtime verification* is concerned with the monitoring of software, providing guarantees that observed executions of a program comply with specified properties. This approach can be used on systems of a complexity that is difficult to address by *static verification* such as systems with numerous interacting sub-units, heavy usage of mainstream libraries, and real world deployments. On the other hand, with runtime verification it is not possible to extrapolate about all possible execution paths. Furthermore, monitoring incurs runtime overheads which may be prohibitive in certain systems.

* Published in *The 20th International Symposium of Formal Methods (FM'15), vol. 9109 of LNCS, pp. 108-125, 2015. Springer.*

In this paper we present a way of addressing these issues by combining runtime verification with static verification. We start by statically verifying the system against a specification, identifying parts which can either be verified automatically or partially resolved, thus leaving a simpler specification to check at runtime, in turn reducing the overheads induced by monitoring.

As observed, static and dynamic verification have largely disjoint strengths — whereas the former excels in data-oriented properties and struggles to handle complex control-flow logic, the latter handles control-flow properties with substantially lower overheads than data-oriented ones. Combining the two approaches can thus allow the verification process to deal with richer properties with greater ease. However, one of the challenges is to identify a specification notation in which properties which refer to both the data- and control-flow of a system can not only be expressed, but also decomposed to ensure applicability of the different verification techniques. In order to address this issue we have, in a previous paper [3], proposed the StaRVOOrS framework. One key part of that framework was the proposal of a specification notation, called *ppDATE*, and a verification methodology, to specify and verify both *control-oriented* properties and *data-oriented* properties.

Our contributions are: i) A formal definition of *ppDATE* (Sec. 3.3); ii) An algorithm to translate *ppDATE* into *DATE* [10], the formalism used in the runtime verification tool Larva [11] (Sec. 3.4); iii) Application of our approach to Mondex [21], an electronic purse application (Sec 4); iv) A description of the results of the case study including an analysis of the verification process providing evidence that our approach substantially reduces the overhead of the runtime monitoring (Sec. 5).

2 The StaRVOOrS Framework

The StaRVOOrS framework (STAtic and Runtime Verification of Object-ORiented Software), which we originally suggested in [3], combines the use of the deductive source code verifier KeY [7] with that of the runtime monitoring tool Larva [11]. KeY is a deductive verification system for data-centric *functional correctness* properties of Java source code, which generates, from JML and Java, proof obligations in *dynamic logic* (a modal logic for reasoning about programs) and attempts to prove them. Larva (*Logical Automata for Runtime Verification and Analysis*) [11] is an automata-based Runtime Verification tool for Java programs which automatically generates a runtime monitor from a property using an automaton-based specification notation *DATE*. Larva transforms the specification into monitoring code together with AspectJ code to link the system with the monitors.

Fig. 1 gives an abstract view of the framework workflow. Given a Java program P and a specification S of the properties to be verified (given in the language *ppDATE*, see Sec. 3), these are transformed into suitable input for the *Deductive Verifier* module which, in principle, might statically fully verify the properties related to pre/post-conditions. What is not proved statically will then be left to

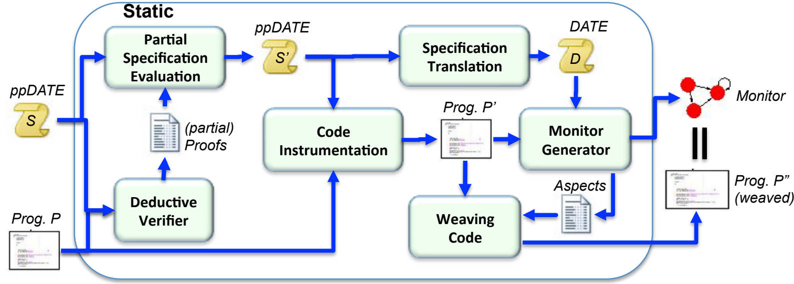


Fig. 1. High-level description of the StaRVOOrS framework workflow

be checked at runtime. Here, not only the completed but also the *partial* proofs are used to generate path conditions for not statically verified executions. The *Partial Specification Evaluator* module then rewrites the original specification S into S', refining the original pre-conditions with the aforementioned path conditions. Note that S' is no longer a full specification of the desired behaviour. Instead, it only specifies executions that are not covered by the static verification.

In a next step, the resulting *ppDATE* specification S' is, via *Specification Translation*, turned into a specification in *DATE* format (D), suitable for the runtime verifier. As *DATE* has no native support for pre/post-conditions, these are simulated by pure *DATE* concepts (see Sec. 3.4). This also requires changes to the code base (done by the *Code Instrumentation* module), like adding counters to distinguish different executions of the same code unit, or adding methods which operationalise pre/post-condition evaluation. The instrumented program P' and the *DATE* specification D are given to the *Monitor Generator*, which uses aspect-oriented programming techniques to capture relevant system events. Later on, the generated aspects are weaved (*Weaving Code*) into P'. The final step in the workflow is the actual runtime verification, which executes the weaved program P'' — running the original program in parallel with a monitor of the simplified property. In case of a runtime error, a trace is produced to be analysed.

3 *ppDATE*: A Specification Language for Data- and Control-oriented Properties

In general, formalisms for specifying software fall into two very different categories. *Data-oriented* properties may be written in expressive formalisms (like first-order logic), but they only talk about specific points in the execution, rather than traces. One instance of this is the *Java Modelling Language* (JML) [16], which mainly allows for the specification of pre/post-conditions of method calls, and class invariants. Such formalisms are not well suited for specifying legal sequences of events or states. The other category, which we call *control-oriented*, offers great flexibility in specifying legal sequences of events or states, using automata or

temporal logics, but supports only simple constraints on data. One instance of this is *Dynamic Automata with Timers and Events* (*DATE*) [10].

In real scenarios, there is often a need to specify both, rich data constraints and legal execution sequences. Still, when formalising such scenarios, traditionally a formalism from either of the two categories is chosen. This leads to *coding* the aspects that are less supported in the respective formalism, like, e.g., coding legal execution traces via model/ghost fields in JML, or coding richer data constraints in *DATE* by extending the code basis with checkers for specific constraints. Even if such codings might be necessary somewhere in the process, we claim that they should not be the duty of the user when formulating properties. Instead, we propose a language which natively supports both types of properties, and let the machinery do necessary codings automatically (as is performed in the modules *Specification Translation* and *Code Instrumentation* in Fig. 1). The language we propose combines features from *DATE* and (very basic) JML, and is called *ppDATE* (*p*re/*p*ost *DATE*), which allows to annotate states with Hoare triples. This also enables us to employ two verification tools in the workflow: KeY, which offers static verification of Java source code annotated with JML, and Larva, which supports runtime verification of *DATE* properties.

3.1 Events

Both *DATE*s and *ppDATE*s use system events to trigger transitions, which typically correspond to the entering or leaving a method or a code block.

Definition 1. *Given an alphabet Σ of named code (typically the union of named functions Φ and named code blocks Λ), we will denote the event marking the entry into $\sigma \in \Sigma$ as $\sigma^{\downarrow}$, and the exit as $\sigma^{\uparrow}$. The set of all such events over alphabet Σ will be written as $\Sigma^{\updownarrow}$.*

For instance, in Java, Φ are methods, and Λ are labelled statements — a singleton statement, whether elementary or structured, can be labelled directly, whereas a sequence of statements, to be named, is put into a labelled block. In addition, we will assume that the system events are indexed by an identifier unique to each execution of a function or block, as in $\sigma^{\downarrow}_{id}$ and $\sigma^{\uparrow}_{id}$. These identifiers can be created automatically using techniques as those presented in [13] or through stack frame references.

3.2 DATE

DATE [10] is an automaton-based control-flow formalism used in Larva. At their simplest level, *DATE*s are finite state automata, whose transitions are triggered by system events (primarily entry points and exit points of methods) and timers, but augmented with a symbolic state which may be used in conditions guarding transitions and can be modified via actions also specified as part of the transition.

As an example of a *DATE*, consider the automaton depicted in Fig. 2, but ignoring the information given in the states. Transitions are tagged as $e \mid c \mapsto a$,

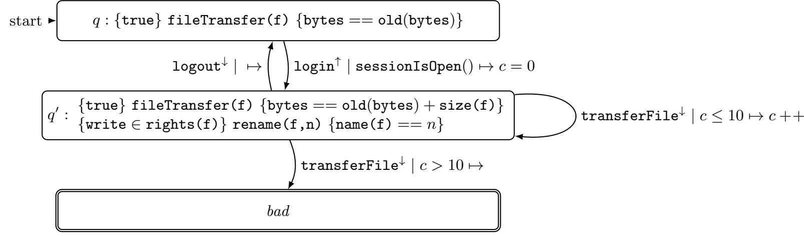


Fig. 2. A *ppDATE* limiting file transfers

where e is the event which triggers the transition, c is the condition which has to hold when event e happens for the transition to be taken, and a is an action to be executed upon taking the transition. Some states (one in the example) are marked as bad states, which indicate that a property violation has taken place when they are reached. The *DATE* component of the property shown (i.e. everything in the diagram except for the information in the states of the automaton) in the example ensures that no more than 10 file transfers take place in a single login session. Note that the specification also uses a new variable as part of the monitor (variable c) which keeps count of the number of files transferred in a single session.

*DATE*s may refer to *valuations* θ of *program variables*. In addition, they also feature another type of variables, called *monitor variables* which do not belong to the program under scrutiny, but instead are local to an automaton, and can be used, for instance, for counting visits to a state (among others). The values of those variables are stored in *valuations* ν of *monitor variables*, and changed only in actions a of transitions. Both actions and conditions in transitions can depend on program variables as well as on monitor variables. Given a condition c, we write $(\theta, \nu) \models c$ to denote that c is satisfied by valuations θ and ν. In the following, Θ denotes the set of all valuations of program variables for a given program under scrutiny.

Definition 2. *A DATE M on a system with program variable valuations over Θ is a tuple $\langle Q, \mathcal{V}, \Sigma, t, B, q_0, \nu_0 \rangle$:*

- *Q is the set of automaton states.*
- *$\mathcal{V}$ is the set of valuations of monitor variables.*
- *Σ is the alphabet, made up of function names Φ and block names Λ.*
- *t is the transition relation among states in Q, where each transition is tagged with (i) the event in $\Sigma^{\updownarrow}$ which will trigger it; (ii) a condition on program and monitor variables; (iii) an action which may change the valuation of monitor variables: $t \subseteq Q \times \Sigma^{\updownarrow} \times \mathcal{P}(\Theta \times \mathcal{V}) \times (\Theta \times \mathcal{V} \longrightarrow \mathcal{V}) \times Q$.*
- *$B \subseteq Q$ is the set of bad states.*
- *$q_0 \in Q$ is the initial state.*
- *$\nu_0 \in \mathcal{V}$ is the initial valuation of monitor variables.*

We will write $q \xrightarrow{e|c\mapsto a}_M q'$ to mean that $(q,e,c,a,q') \in t$. The subscript M is omitted if it is clear from the context. We say that a DATE is deterministic whenever the following hold: if $q \xrightarrow{e|c\mapsto a} q'$ and $q \xrightarrow{e|c'\mapsto a'} q''$ and $q' \neq q''$, then c and c' are mutually exclusive, i.e. $c \cap c' = \emptyset$.

Consider once again, the *DATE* shown in Fig. 2. This can be formalised as follows: $M = \langle Q, \mathcal{V}, \Sigma, t, B, q_0, \nu_0 \rangle$ over program variable valuations Θ, where: $Q = \{q, q', \textit{bad}\}$, $\mathcal{V} = \{(c,n) \mid n \in \mathbb{Z}\}$, $\Sigma = \{\texttt{fileTransfer}, \texttt{login}, \texttt{logout}\}$, $B = \{\textit{bad}\}$, $q_0 = q$, $\nu_0 = (c, 0)$. Furthermore, the transition relation t consists of four elements, including $q' \xrightarrow{\texttt{fileTransfer}^{\downarrow}|c \leq 10 \mapsto c{+}{+}} q'$ and $q' \xrightarrow{\texttt{fileTransfer}^{\downarrow}|c>10 \mapsto \texttt{skip}} \textit{bad}$.

We can now define the semantics of *DATE*s by identifying how a trace generated by the system changes the states of the *DATE*.

Definition 3. *We define that a* trace $w \in (\Sigma^{\updownarrow} \times \Theta)^*$ shifts a monitor from configuration $(q,\nu) \in Q \times \mathcal{V}$ to configuration $(q',\nu') \in Q \times \mathcal{V}$, *written* $(q,\nu) \stackrel{w}{\Rightarrow} (q',\nu')$, *by induction over w:*

$$(q,\nu) \stackrel{\varepsilon}{\Rightarrow} (q',\nu') \stackrel{df}{=} q = q' \wedge \nu = \nu';$$
$$(q,\nu) \xRightarrow{(e,\theta):w} (q',\nu') \stackrel{df}{=} \exists\, q'', \nu'' \;\cdot\; \exists\, c, a \cdot$$
$$q \xrightarrow{e|c\mapsto a} q'' \wedge ((\theta,\nu) \models c) \wedge \nu'' = a(\theta,\nu) \wedge (q'',\nu'') \stackrel{w}{\Rightarrow} (q',\nu');$$
$$(q,\nu) \xRightarrow{(e,\theta):w} (q',\nu') \stackrel{df}{=} (q,\nu) \stackrel{w}{\Rightarrow} (q',\nu') \wedge \nexists\, q'', c, a \;\cdot\; q \xrightarrow{e|c\mapsto a} q'' \wedge ((\theta,\nu) \models c).$$

Given a DATE M, a trace $w \in (\Sigma^{\updownarrow} \times \Theta)^*$ is said to be a counter example *if both $(q_0,\nu_0) \stackrel{w}{\Rightarrow} (q,\nu)$ and $q \in B$.*

The set of *violating traces of a DATE M*, written $\mathcal{VT}(M)$ is defined to be traces which have a counter example of M as a prefix.

What we have given here is a subset of the full expressive power of *DATE*s. *DATE*s support further features, including: (i) timers which may be used in the transition conditions or as events to trigger transitions; (ii) communication between *DATE* automata using standard CCS-like channels with $c!$ acting as a broadcast on channel c and which can be read by another automaton matching on event $c?$; and (iii) replication of automata through which every time a particular event in some way distinct from earlier ones (e.g. using a method's parameters or the target object) is received a new automaton is created (e.g. used to replicate a property for each instance of a class). We use the latter two features of *DATE*s when translating *ppDATE*s into *DATE*s. Refer to [10] for the semantics of *DATE*s.

3.3 ppDATE

ppDATE extends *DATE* with elements of data-oriented specification, by assigning (zero or more) Hoare triples to each state. Intuitively, upon entering the code unit $\sigma \in \Sigma$ while in a state which contains a Hoare triple $\{\pi\}\sigma\{\pi'\}$, and given

that pre-condition π was satisfied, one should ensure that post-condition π' is satisfied upon exit of σ.

Let us reconsider the property shown in Fig. 2, this time also looking at the information given in the states. Some states are tagged with Hoare triples which should hold when the automaton lies in that state. In addition to ensuring no more than 10 transfers per login session, the Hoare triples also ensure that: (i) the number of bytes transferred increases when a file transfer is done while logged in, (ii) but not when an attempt to transfer a file is done when logged out; and (iii) renaming a file works as expected if the user has the sufficient rights and is logged in.

In *ppDATE*s pre/post-conditions are evaluated over *valuations* θ of *program variables* (defined as for *DATE*s, cf. Sec. 3.2). For instance, $\theta \models \pi$ may or may not hold, where θ is a mapping from program locations like object fields, array fields, and method parameters, to values of the right type.

Definition 4. *A ppDATE (pre/post-condition DATE) M_p on a system with program variable valuations over Θ consist of (i) a DATE $M = \langle Q, \mathcal{V}, \Sigma, t, B, q_0, \nu_0 \rangle$ and (ii) a function τ which tags each state of the automaton with Hoare triples for particular function and block names: $\tau \in Q \longrightarrow \mathcal{P}(\mathcal{P}(Q) \times \Sigma \times \mathcal{P}(Q))$.*

Notation for transitions, and definition of configuration changes over strings of system behaviour are carried over unchanged from *DATE*s. We use the usual Hoare triple notation $\{\pi\}\,\sigma\,\{\pi'\} \in \tau(q)$ to denote $(\pi, \sigma, \pi') \in \tau(q)$. Although determinism on the Hoare triples' preconditions is not problematic in itself, we choose to extend the determinism condition to ensure that for any two Hoare triples in a single state over the same function have disjoint precondition so as to have a more effective monitoring algorithm of these triples: for any $\{\pi_1\}\,\sigma\,\{\pi_1'\}$ and $\{\pi_2\}\,\sigma\,\{\pi_2'\}$ in $\tau(q)$, $\pi_1 \cap \pi_2 = \emptyset$.

To formalise the *ppDATE* shown in Fig. 2 we use the *DATE* defined earlier, and add a function τ mapping states to sets of Hoare triples, such as:

$$\begin{aligned}\tau(q') = \{\ & \{\texttt{true}\}\ \texttt{fileTransfer(f)}\ \{\texttt{bytes} == \texttt{old(bytes)} + \texttt{size(f)}\},\\ & \{\texttt{write} \in \texttt{rights(f)}\}\ \texttt{rename(f,n)}\ \{\texttt{name(f)} == \texttt{n}\} \quad \}\end{aligned}$$

We can now define the semantics of *ppDATE*s by extending the notion of counter-examples to include violations of postconditions.

Definition 5. *Given a ppDATE $M_p = \langle M, \tau \rangle$, a trace $w \in (\Sigma^{\updownarrow} \times \Theta)^*$ is said to be a* counter example *if either (i) w is a counter example of M; or (ii) w can be decomposed into four parts $w = w_1 +\!\!+ \langle(\sigma_{id}^{\downarrow}, \theta_1)\rangle +\!\!+ w_2 +\!\!+ \langle(\sigma_{id}^{\uparrow}, \theta_2)\rangle$ such that the following conditions hold:*

(a) Trace w_1 takes M from the initial configuration to some configuration (q, ν): $(q_0, \nu_0) \stackrel{w_1}{\Longrightarrow} (q, \nu)$;
(b) There is a Hoare triple of σ enforced in state q: $\{\pi\}\,\sigma\,\{\pi'\} \in \tau(q)$;
(c) Valuation θ_1 satisfies the precondition: $\theta_1 \models \pi$;
(d) Valuation θ_2 does not satisfy the postcondition: $\theta_2 \not\models \pi'$.

Recall that each event in a trace is annotated with an identifier, unique per entry-exit pair — therefore, the $\sigma^{\downarrow}_{id}$ and $\sigma^{\uparrow}_{id}$ appearing in the trace (i) match the method named σ in the Hoare triples; and (ii) ensures (by construction of the identifiers) that σ_{id} does not appear in w_2.

As before, the set of *violating traces of a ppDATE* M_p, written $\mathcal{VT}(M_P)$ is defined to be traces which have a counter example of M_P as a prefix.

Note that Definition 5 allows for the inclusion of events corresponding to calls to the methods specified in the states (part of the Hoare triples). This is natural as concrete traces in *ppDATE*s do not necessarily coincide with "paths" of the DATE component of the *ppDATE*. For instance, in Fig. 2 a call to `rename(·)` when in state q' is a valid one and a corresponding event $\texttt{rename}^{\downarrow}$ will be present in the trace of the *ppDATE*.

3.4 Translation from *ppDATE* to *DATE*

In our architecture, KeY first tries to prove all data-oriented parts of a *ppDATE* S, and the partial proofs are used to get an optimised *ppDATE* S'. To make the property S' runtime-checkable, we further translate away the (remaining/optimised) Hoare triples, to arrive at a set of parallel[4], pure *DATE*s D that can be processed by Larva. One complication in the translation is the possibility that a Hoare triple in a state may 'clash' with an outgoing event. This would for instance be the case if we added to Fig. 2 a transition from q to q' with $\texttt{fileTransfer}^{\downarrow}$ as a triggering event. For clarity of presentation we give two algorithms, one for the case when no such clashes arise, and then for the full case. Formally, we define a clashing Hoare triple as follows.

Definition 6. *Given a ppDATE $M_p = \langle M, \tau \rangle$ with $M = \langle Q, \mathcal{V}, \Sigma, t, q_0, \nu_0 \rangle$, a Hoare triple $\{\pi\}\,\sigma\,\{\pi'\} \in \tau(q)$, for some $q \in Q$, is called* clashing *if an outgoing transition from q is guarded by event $\sigma^{\downarrow}$ (i.e., $\exists\ c, a, q' \ \cdot\ q \xrightarrow{\sigma^{\downarrow}|c \mapsto a} q'$). A* clash-free *ppDATE is a ppDATE with no clashing Hoare triple.*

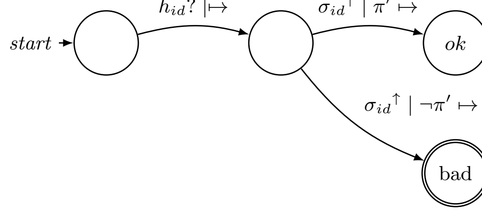
We now present the algorithm to translate a clash-free *ppDATE* into *DATE*s. The translation works by replacing each Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ in a state q by a new reflexive transition (from q to q) triggered by an entry into function σ such that the precondition π holds, and sending a message which is used to replicate a parallel post-condition checking *DATE* automaton.

Algorithm 1 *Given a* clash-free *ppDATE $M_p = \langle M, \tau \rangle$, we can construct a set of parallel DATEs equivalent to M_p in the following manner:*

1. *Give each Hoare triple in M_p a unique name h, to be interpreted as a channel name in the DATEs to be constructed.*

[4] Multiple, parallel *DATE*s define behaviour of a sequential application in the sense that each event in the application may trigger transitions in a number of *DATE*s. In addition, the *DATE*s can synchronise with each other by means of channels.

2. *For each Hoare triple h, construct a replicated DATE automaton M_h (called* the post-condition checker*), parameterised over identifier id, as shown below*[5]*:*



3. *Turn M, the DATE component of M_p, into the DATE M' such that, for each Hoare triple $\{\pi\}\,\sigma\,\{\pi'\}$ named h in $\tau(q)$ is replaced by a transition $q \xrightarrow{\sigma_{id}{}^{\downarrow} \mid \pi \mapsto h_{id}!}_{M'} q$.*
4. *The resulting set of parallel DATEs is defined to be:* $\{M'\} \cup \{M_h \mid h$ *is a Hoare triple identifier from* $M\}$.

This translation works well except that it would introduce non-determinism when the *ppDATE* includes clashes. To extend the translation to work in the presence of clashes, we transform Hoare triples clashing with a transition into a family of disjoint transitions each of which performs the transition but also checks whether the post-condition checker should be triggered.

Algorithm 2 *Given a general ppDATE M_p, we can construct a set of parallel DATEs equivalent to M_p by following Algorithm 1 except that we replace Step 3., by the following rule:*

3a. Each non-clashing Hoare triple h: $\{\pi\}\,\sigma\,\{\pi'\}$ in $\tau(q)$ is turned into a transition $q \xrightarrow{\sigma_{id}{}^{\downarrow} \mid \pi \mapsto h_{id}!}_{M'} q$.

3b. For each clashing Hoare triple h: $\{\pi\}\,\sigma\,\{\pi'\} \in \tau(q)$, clashing with n outgoing transitions, $q \xrightarrow{\sigma^{\downarrow} \mid c_i \mapsto a_i} q_i$ $(0 \leq i < n)$:

- *Replace $q \xrightarrow{\sigma^{\downarrow} \mid c_i \mapsto a_i}_M q_i$ with $q \xrightarrow{\sigma_{id}{}^{\downarrow} \mid c_i \mapsto \{a_i;\, \mathit{if}\, \pi\, \mathtt{then}\, h_{id}!\}}_{M'} q_i$;*
- *Add the following transition $q \xrightarrow{\sigma_{id}{}^{\downarrow} \mid (\neg c_0 \wedge \ldots \wedge \neg c_n \wedge \pi) \mapsto h_{id}!}_{M'} q$.*

4 Case study: Mondex

Mondex is an electronic purse application used by smart cards products [1], and it has been used as a benchmark problem within the Verified Software Grand Challenge context [21]. Mondex's original sanitised specification written in Z, together with hand-written proofs of different properties, can be found in [17]. Our variant is strongly inspired by a JML formalisation given in [19]. However, *ppDATE* has native (automata) states, unlike Z or JML. This allowed us to naturally represent the overall status of the observed system by states (see the

[5] Following the semantics of *DATE*s, whenever a message is received on channel h with a new identifier, this automaton is replicated and the first transition is taken.

nodes of the graph in Fig. 3), instead of representing the status by additional data like in Z or JML.

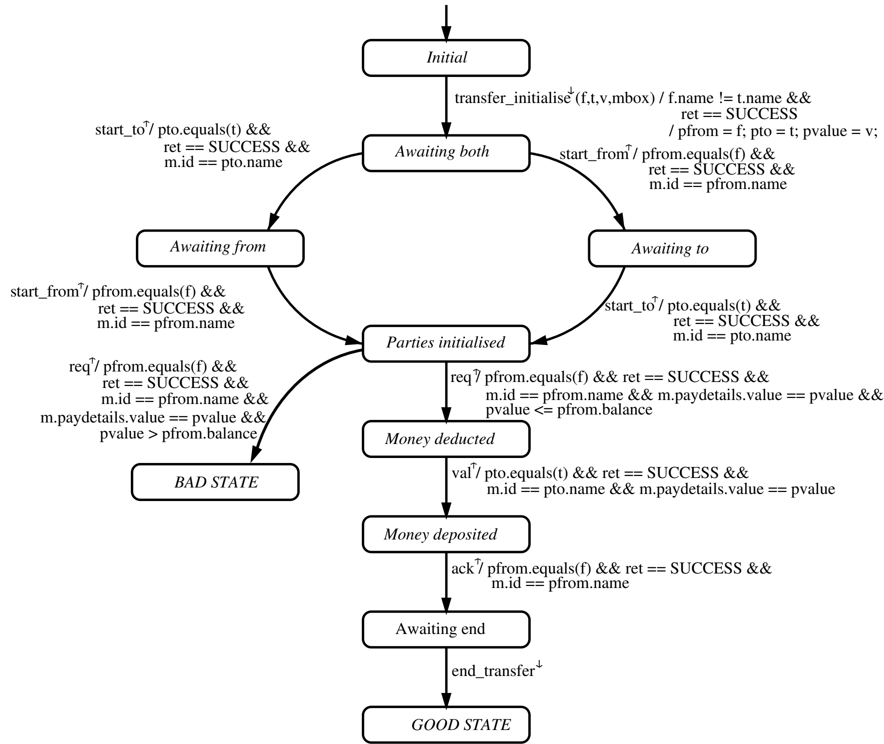
Mondex essentially provides a financial transaction system supporting transferring of funds between accounts, or 'purses'. We focus on analysing the transactions taking place between these purses, which follow a multi-step message exchange protocol: whenever a transaction between two purses is to take place, (i) the source and destination purses should (independently) register with the central fund transferring manager; (ii) then a request to deduct funds from the source purse may arrive, followed by (iii) a request to add the funds to the destination purse; and (iv) finally, there should be an acknowledgement that the transfer took place, before the transaction ends.

The original version of Mondex works on Java Card, and all controls in relation to security properties have to be handled on the card, rather than being monitored on an external source. In our case study, we have verified a version of the protocol which works using a server, rather than a smart card. The only principal difference in the protocol implementation is that the server version uses return values to control the protocol rather than raising Java Card exceptions. The full specification and code of the case study can be found in [2]. The specification consists on a *ppDATE* automaton with 10 states, 25 transitions and a total of 26 different Hoare triples. The implementation consists on 514 lines of code (without comments) which are distributed over 8 files.

4.1 *ppDATE* Property

As typical in transaction-based systems, the Mondex case study illustrates how complexity can arise from accessing different purses concurrently and in a manner not predicted by the system developer. Specifications of such systems have to reflect this emerging complexity and include (i) constraints as to the control flow of the system — the order in which different components are accessed; and (ii) constraints on how these components behave when accessed both when access is expected and when it is not. Our formalism, *ppDATE*, addresses both these orthogonal issues in a structured manner.

The top-level specification of the Mondex purse-management systems can be found in Fig. 3. For space reasons, the Hoare triples populating the states are not depicted in this figure, but instead, we will show them for specific states further on. At the automaton level, the *ppDATE* (which we will call S) expresses the protocol governing how the purses are to be accessed, by specifying the order in which the components (in this case methods used to access the purses) can be called. For instance, after the parties are initialised (encoded in S's state named Parties Initialised), a request to deduct more money than what is found in the source purse should fail, while a request to deduct an amount of money which is available should take us to a state (named Money deducted) in which the protocol now allows for the money to be transferred to the destination purse. The ordering is crucial and appears in practically all financial transaction systems so as to ensure that no money will be created at any point in the transaction.



In addition:
* All states have outgoing transitions for ret == SUCCESS && SENDER!=party (where party is the party from whom a message is not expected) going to a bad state.
* All states but 'Awaiting end' have outgoing transitions for end_transfer, going to a bad state.

Fig. 3. *ppDATE* to monitor the behaviour of the transaction protocol.

Similarly, access to any unregistered purse takes us to a bad state[6] since the system should never allow these methods to be accessed. Notice that comparing `m.paydetails.value` and `pvalue` is needed in order to check that the message received is part of the ongoing transaction.

Over and above the specification of the protocol, one has to specify the behaviour of the involved methods, which obviously changes together with the status of the protocol. For instance, transfer of funds from a purse to another should succeed once both purses have been registered, but should fail if attempted before registration or if an attempt is made to perform the transfer multiple times. This behaviour is encoded by different Hoare triples assigned to different S

[6] These transitions are not drawn in the diagram (but are mentioned in the note underneath) so as to avoid confusion. Note that Larva will not take any explicit action when reaching a bad state: the corresponding automaton will stay in that state until the whole monitor is restarted (unless it is explicitly specified on the monitor what action to take when reaching a bad state). A log is kept indicating this.

states. For instance, just after the registration of two purses (in S's state Parties initialised), the method `val_operation` which requests money from the source purse should succeed and deduct the money from the purse (provided enough money is available) as shown in the Hoare triple[7] below:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance); }
val_operation
{ \result == SUCCESS
  && (balance == \old(balance) + transaction.value); }
```

On the other hand, if the same method is accessed after the funds have already been deducted (S's state Money deducted) then the purse content should remain unchanged, and the request should be ignored:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance); }
val_operation
{ \result == IGNORED; }
```

Note that both Hoare triples above have the same pre-condition, but the different *ppDATE* states they belong to demand different behaviour (i.e., post-conditions) of `val_operation`.

The control-oriented properties basically ensure that the message exchange goes as expected. In contrast, the pre/post-conditions (in total, there are 26 Hoare triples in the states of the *ppDATE*) ensure the well-behaviour of the individual steps.

4.2 Combined Static and Runtime Verification

Following the verification approach from Fig. 1, we start by extracting the Hoare triples from the *ppDATE* which are translated to JML annotations in the source code. KeY then generates corresponding proof obligations in dynamic logic and starts a proof attempt. Note that, in this work, we use KeY only fully automatically, not using its rich support for interactive theorem proving, neither assuming user provided proof-hinting annotations (like loop invariants).

When trying to prove these formulae, KeY creates proof branches corresponding to case distinctions in the code. Usually, KeY manages to automatically close the proofs of the simpler branches, but may not (automatically) close more difficult branches. Still, the open goals contain path conditions, i.e., conditions on the valuation of program variables *before* the method was entered. We use this information to refine the pre-condition to the cases where KeY cannot close the proof.

For instance, consider the part of the specification already discussed in the previous section — the JML pre/post-condition from the the state Parties initialised, when a request for a money transfer is received:

```
requires checkSameTransaction() == SUCCESS
         && transaction.value <= (ShortMaxValue - balance);
```

[7] In the pre- and post-conditions, we use basic JML expression syntax [16].

```
ensures \result == SUCCESS
        && (balance == \old(balance) + transaction.value);
```

The code (and consequently KeY) branches on the status of the transaction, and one of the branches, when the transaction is not awaiting a money deduction request, is closed successfully. The other branch is left open. From the open goal, we can read off the path condition `status == ProtocolStatus.Epv` (i.e. the receiver purse is expecting to receive the requested value). Only if this condition holds upon entry of `val_operation`, the post-condition will need to be checked at runtime. All other cases are proved correct statically, by KeY. Before generating the runtime monitor, we therefore refine the corresponding Hoare triple in *ppDATE* to include this path condition:

```
{ checkSameTransaction() == SUCCESS
  && transaction.value <= (ShortMaxValue - balance)
  && status == ProtocolStatus.Epv; }
val_operation
{ \result == SUCCESS
        && (balance == \old(balance) + transaction.value); }
```

In our case study, except for two Hoare triples related to the initialisation and termination of a transaction which were fully proven by KeY, all the other 24 triples were refined in this manner.

The resulting *ppDATE* specification can now be transformed into an equivalent *DATE*, and the runtime verification tool LARVA is used to monitor the system for possible violations.

The implementation of Mondex we describe in this section has been fully verified with our technique, albeit in an iterative manner since verification revealed some errors we made in our original implementation of Mondex (see next section).

5 Experimentation

Here, we summarise the experimental results of applying our approach to the Mondex case study. In particular, we compare execution times of (a) the unmonitored implementation, (b) the monitored implementation using the original specification S, and (c) the monitored implementation using specification S', obtained from S via static (partial) proof analysis. The table below shows the average execution time, on a PC Intel Core i7 using a single core, for these three scenarios when the system is ran performing different numbers of transactions. Statically analysing all the Hoare triples took KeY around 2.15 minutes. However, the real gain is that this analysis is done once and for all prior to deployment, and the gains reported in the table below improves performance for all executions once the system is deployed.

Transactions	(a) no monitoring	(b) monitoring S	(c) monitoring S'
10	8 ms	120 ms	15 ms
100	50 ms	3500 ms	90 ms
1000	250 ms	330000 ms	375 ms

As expected, adding a monitor caused overhead on the execution time (b). However, this overhead is substantially reduced by using our approach (c). The relative difference is quite remarkable: at least 10 times faster for low number of transactions, and increasing up to 900 times faster as the number of transactions increases. This large reduction in execution time overheads when optimising the monitor is primarily due to the fact that data-oriented monitoring can be prohibitively expensive in the first place. In fact, using our approach, each function with a satisfied precondition fires an additional automaton being traversed in parallel. This results in the large overheads in the case study. However, by pruning away many of these checks through the typical case of a strengthening of the precondition results in the gains we obtain. This indicates that using static analysis to pare down the data-oriented aspect of the properties is ideal in this situation, in that we are attacking directly the overlap between a strength of static analysis and a weakness of dynamic analysis.

Note that it is usually impossible to get a full proof when using a static verifier like KeY in the simple way we do, i.e., without user interaction, and without poof supporting annotations (like loop invariants). But the missing proving power is only one aspect. The other is that branches may be open because the corresponding execution path is actually erroneous. KeY cannot *per se* distinguish these two cases, but Larva can detect the erroneous case when it appears at runtime. Note that the above table does not say anything about errors revealed by applying our approach to the case study. It only shows execution times of the various scenarios *after* errors were revealed and removed. However, finding errors is the one of the most important purposes of verification, so we briefly discuss some errors in the following.

In our variant of Mondex, in order to scale the transaction count, several purses are iteratively generated, using the index for the name of the purse created in each iteration. Executing the application with the monitor generated by StaRVOOrS led to a runtime failure. Inspecting the monitor-generated (failing) execution trace allowed us to spot the problem. Originally, the index of the loop was initialised with 0, but the names of the purses were assumed to be greater than 0. This lead to a purse with an invalid name, causing a failure which was detected by StaRVOOrS.

We have also intentionally injected errors into the Mondex case study, to test whether the approach would detect them. All of them were successfully detected with StaRVOOrS. We have also considered incomplete or wrong specifications. This can mean very different things. In a case where the specification is too weak, such that the implementation fulfils it for wrong reasons, we may not catch that. This is a common issue for practically all verification approaches. At least, in our approach of combined data- and control-oriented verification, we have some chance that a problem propagates to a state where the specification is strong enough to catch it. If on the other hand a Hoare triple accidentally puts wrong demands on the implementation, KeY will naturally not be able to prove it. Thereby, the StaRVOOrS methodology ensures that the property is checked at runtime. Even there, verification will fail (if only that part is executed),

but this time, we get a failing trace. Analysing it will show that, actually, the computation was fine, suggesting that the specification was wrong in the first place. For example, the post-condition used for static analysis (see Sec. 4.2) of the method initialising the sender purse during a transaction used a wrong variable, and KeY was not able to prove it. At runtime, the replicated automaton checking the post-condition shifted to a bad state, even if the computation lead to the expected results, allowing us to spot, and correct, the failing post-condition.

6 Related Work

The combination of different verification techniques is gaining more and more popularity. One active area of research is the combination of static analysis and testing, e.g. [4, 9, 12, 14, 15, 18]. A direct comparison of our work with those would not be fully fair as we have different objectives. We are not aiming at generating test cases, but at monitoring the actual post-deployment runs of the system. What we have in common is that static analysis/verification is used to limit the dynamic efforts, there by filtering test cases, here by filtering checks at runtime.

A different line of research is the combination of testing and runtime verification, as done by Falzon and Pace in [13] where QuickCheck and Larva are combined. Similar to *ppDATE*s, QuickCheck automata employ pre-postconditions but as part of the transitions as opposed to the states as used in *ppDATE*s.

The work by Wonisch *et al.* in [20] is concerned with the use of program transformation to avoid unsafe program executions. Their main objective is the optimisation of runtime monitoring by using static analysis (rather than full-blown static *verification*) techniques.

In [8] static analysis is used to improve the performance of runtime monitoring based on tracematches. The paper presents a static analysis technique to speed up trace matching by reducing the runtime instrumentation needed. The static analysis part is based on three stages: ruling out some tracematches, eliminating inconsistent instrumentation points, and finally further refinement of the analysis taking into account execution order.

In [22], Zee *et al.* explore the combination of static and runtime verification, aiming at a specification language whose specifications can be both statically analysed and runtime checked. They extend the static verifier Jahob with techniques to verify specifications at runtime, and can execute specifications using quantifiers, set comprehensions, integer and object expressions amongst other constructs. Most of the properties they can verify are data-centric, whereas we also cover control-centric properties. We could benefit from incorporating some of their solutions for complex data structures in our approach.

Several specification approaches, like SPARK [5], JML and SPEC# [6] are supported by both static and runtime verification tools. However, to the best of our knowledge, static verification is not used for optimising runtime verification.

7 Conclusions

In this paper we have presented the StaRVOOrS framework combining (partial) static and (optimised) runtime verification. As a first step, we have instantiated our approach with the tools KeY and Larva. We have presented and formalised a notation, *ppDATE*, which allows us to arbitrarily combine control-oriented (based on automata with event-triggered transitions) and data-oriented (relating final and initial data values) properties in a single formalism, and thereby to describe a larger variety of applications. An additional interesting aspect of this combination is that data-oriented properties formulated in a pre/post style can be made dependent on the history of previous events.

To illustrate how this framework works, we have applied it to a variant of the Mondex case study [19, 21]. In this case study, we analyse the behaviour of the transaction protocol for transferring money between electronic purses, and we demonstrate how this protocol can be partly statically, partly dynamically verified using our framework. Apart from this case study, we have also applied our framework on a different case study — a simple system, in which users may login and perform different operations (see [2] for the sources of this case study).

The difference in performance between the fully monitored and the version with simplified monitors is, in itself, motivation to look further into how we can extend our approach. The huge gains are primarily a side-effect of the large costs of data-oriented property monitoring, meaning that any reduction in the magnitude of the monitored properties can lead to large reductions in overheads. Our approach may thus be a way of dealing with this class of properties which one typically shies away from monitoring due to the large overheads involved.

The exact gain of optimising runtime monitoring by static results will vary depending on the application, but in our approach, it will be substantial whenever there are enough paths through the computation which are simple enough for automated (static) verification, and yet appear frequently during runtime, which arguably is common in many applications. In addition, we want to highlight that the combination of static and runtime verification does not only speed up the execution time of a monitored system, but moreover increases confidence, as parts of the system are proved to be correct once and for all.

Both, the efficiency gain for monitoring and the confidence gain, will only increase along with future improvements in the used static verifier. For instance, if ongoing work on loop invariant generation in KeY will lead to closing some more branches in typical proofs, then this will have an immediate effect that is proportional to the frequency of executing those loops at runtime.

We are currently proving the soundness of the transformation of *ppDATE*s to *DATE*s, and automating the verification process to use KeY and Larva with *ppDATE*s. Finally, an interesting question is whether static verification could be used to (partially) prove the control-oriented part of *ppDATE*s. This is an open question left for future work.

Acknowledgements. We would like to thank Christian Colombo and Martin Henschel for their support concerning implementation issues about Larva and

KeY respectively. We also thank the anonymous reviewers for their valuable comments to improve the presentation of the paper.

References

1. MasterCard International Inc. Mondex. `www.mondexusa.com/`.
2. StaRVOOrS. `www.cse.chalmers.se/~chimento/starvoors/files.html`.
3. Wolfgang Ahrendt, Gordon Pace, and Gerardo Schneider. A Unified Approach for Static and Runtime Verification: Framework and Applications. In *ISOLA'12*, LNCS 7609. 2012.
4. Cyrille Artho and Armin Biere. Combined static and dynamic analysis. In *AIOOL'05*, volume 131 of *ENTCS*, pages 3–14, 2005.
5. John Barnes. *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012.
6. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS'05*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
7. Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
8. Eric Bodden, Laurie J. Hendren, and Ondrej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, LNCS 4609, 2007.
9. Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM'12: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 132–146, 2012.
10. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *FMICS'08*, volume 5596 of *LNCS*, pages 135–149. Springer-Verlag, September 2009.
11. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA - A Tool for Runtime Monitoring of Java Programs. In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.
12. Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 422–431, 2005.
13. Kevin Falzon and Gordon Pace. Combining testing and runtime verification techniques. In *Model-based Methodologies for Pervasive and Embedded Software*, volume LNCS 7706, 2012.
14. Cormac Flanagan, K. Rustan M Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In Jens Knoop and Laurie J. Hendren, editors, *PLDI'02*, pages 234–245. ACM, 2002.
15. Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. Dyta: dynamic symbolic execution guided with static verification results. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 992–994, 2011.
16. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual. Draft 1.200*, 2007.

17. Susan Stepney, David Cooper, and Jim Woodcock. An Electronic Purse: Specification, Refinement and Proof. *Technical monograph PRG-126, Oxford University Computing Laboratory*, 2000.
18. Nikolai Tillmann and Jonathan de Halleux. Pex-White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hhnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.
19. Isabel Tonin. Verifying the Mondex case study. The KeY approach. *Technical Report 2007-4, Universität Karlsruhe*, 2007.
20. Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Zero Overhead Runtime Monitoring. In *SEFM'13*, volume 8137 of *LNCS*, pages 244–258. Springer Berlin Heidelberg, 2013.
21. Jim Woodcock. First Steps in the Verified Software Grand Challenge. In *SEW'06*, pages 203–206. IEEE Computer Society, 2006.
22. Karen Zee, Viktor Kuncak, Michael Taylor, and Martin C. Rinard. Runtime Checking for Program Verification. In *RV'07*, volume 4839 of *LNCS*, pages 202–213. Springer, 2007.