Model-Driven Consistency Checking of Behavioural Specifications

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@tudelft.nl

Arie van Deursen
Delft University of Technology and CWI
The Netherlands
arie.vandeursen@tudelft.nl

Abstract

For the development of software intensive systems different types of behavioural specifications are used. Although such specifications should be consistent with respect to each other, this is not always the case in practice. Maintainability problems are the result. In this paper we propose a technique for assessing the consistency of two types behavioural specifications: scenarios and state machines. The technique is based on the generation of state machines from scenarios. We specify the required mapping using model transformations. The use of technologies related to the Model Driven Architecture enables easy integration with widely adopted (UML) tools. We applied our technique to assess the consistency of the behavioural specifications for the embedded software of copiers developed by Océ. Finally, we evaluate the approach and discuss its generalisability and wider applicability.

1. Introduction

System understanding is a prerequisite for modifying a software intensive system [1]. As such the (typical) absence of up-to-date design documentation hampers successful software maintenance and evolution. In this paper we address this problem for the documentation of a system's behaviour. We focus on ensuring the consistency of two types of behavioural specifications: interaction-based and state-based behavioural models. The use of such specifications is illustrated by the development process depicted in Figure 1. It is based on the well-known V-model [2] and the starting point of our research.

On the left branch of the 'V' analysis activities take place. Based on Requirements, the high-level Architecture is defined. This architecture identifies the main components of the system and assigns responsibilities. In parallel requirements are made more concrete by Use cases

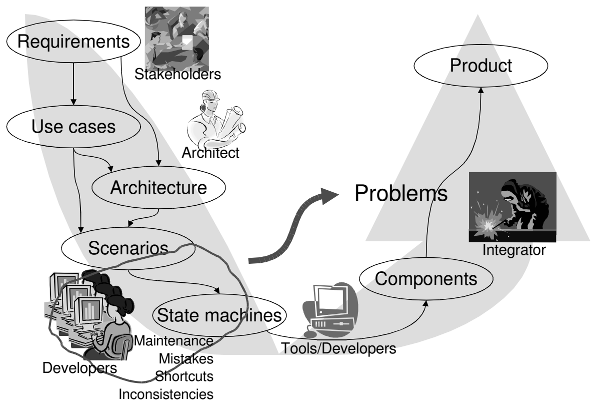


Figure 1. Typical development process

that specify typical interactions a user may have with the system. One distinctive property of use cases is that the system is considered to be a *black box* [3]. These use cases are the first interaction-based behavioural models.

Based on the use cases a set of Scenarios is defined that specifies the interactions of the system's components in terms of exchanged messages. Typically, every use case results in one (normal behaviour) or more (including exceptional behaviour) scenarios. These scenarios are also interaction-based behavioural models, but now the system is considered to be a *white-box*; they show the interactions between the components defined by the architecture.

Eventually, the architecture's components need to be implemented. This requires a complete behavioural specification. Scenarios are, however, not intended to provide such a specification for an individual component. Not only is the specification of a component's behaviour scattered across multiple scenarios, they also are usually only defined for the components' most typical and important behaviours. Therefore, a complete state-based behavioural model, a State machine, is created for each component based on the set of scenarios. This state machine is used to implement or generate the component. Finally, on the right-hand side of the 'V', the different components are integrated into a complete product.

Such a software development process, where state-based component design is based on the specification of a set of use cases, is advocated by many component-based, object-oriented, and real-time software development methods [4–7]. As such, many software development organisations deploy similar development processes.

As software evolves it is often the case that changes are made to 'downstream' software development artefacts without propagating the changes to the corresponding 'upstream' software development artefacts. This can be the result of change requests, but also of design flaws that are only discovered on a more detailed level. Even more inconsistencies are simply introduced by misinterpretations of 'upstream' development artefacts.

In this paper we focus on inconsistencies between interaction-based behavioural models and state-based behavioural models. Inconsistencies between these models can be particularly important because they decompose behaviour along different dimensions. Interaction-based models are decomposed according to the different use cases, that is, they are *requirements*-driven. State-based models, on the other hand, are decomposed according to the different components that were identified during architecture design, that is, they are *architecture*-driven. This makes it hard to discover inconsistencies [8, 9]. Furthermore, when different development groups are responsible for the development of the different architectural components, and these groups individually resolve inconsistencies in different ways, this may obviously lead to problems during integration and maintenance.

In industrial practice behavioural models are often specified as UML models. Moreover, tools are available that, based on UML, are capable of generating source code from such models. Considering such a model-based infrastructure, we believe it makes sense to view consistency checking of behavioural specifications as a model transformation problem. In this paper we investigate what the advantages and disadvantages are of using model transformation technology to discover inconsistencies between interaction-based and state-based behavioural models. Furthermore, we aim to minimise the impact of our approach on existing development processes, for instance, in terms of the languages and tools used.

In Section 2 we introduce the industrial case that motivated this paper: an embedded software control component developed by Océ, a large copier manufacturer. At Océ an important copier subsystem is developed using a process corresponding to Figure 1. Moreover, the components for this subsystem are generated from state machine models. As such, debugging, for instance, is performed on the level of state machines. As a result inconsistencies between scenarios and state machines become even more likely, making it a concern for Océ. Other work on the relation between scenarios and state machines is discussed in Section 3. The enabling technologies for our approach, as well as, the relevant part of the underlying UML specification, and our process for consistency checking are discussed in Section 4. In Section 5 we customise an existing mapping between scenarios and state machines based on Whittle and Schumann [10] for specification as model transformations and consistency checking.

Using our approach we identified several inconsistencies in the behavioural specifications of an industrial system that could lead to integration and maintenance problems. These are discussed in Section 7. Finally, we reflect on our approach in Section 8 and conclude with an overview of the contributions of this paper and opportunities for future work in Section 9.

2. Running Example

Our original motivation for investigating the consistency between interaction- and state-based behavioural models comes from a product-line architecture for embedded software in copiers developed by Océ. We use this architecture as our running example and case study, and for that reason briefly explain it first.

At Océ a reference architecture for copier *engines* is developed. In a copier both the scanning and printing subsystems are referred to as an engine. The reference architecture describes an abstract engine that can be instantiated for (potentially) any Océ copier.

As a running example we use one of the reference architecture's components: the Engine Status Manager (ESM). This component is responsible for handling status requests and status updates in the engine. ESM and the other main components of the reference architecture are depicted in Figure 2.

In a copier engine ESM communicates with two types of components: status control Clients, and Functions. Clients request engine state transitions. Requests by the external status control client (Controller) are translated by the EAI (Engine Adapter Interface) component. To perform status requests of Clients, ESM controls the status of individual Function components. Functions, in turn, recursively control the status of their composing Functions.

For the development of ESM and other engine components a process is used similar to the process outlined

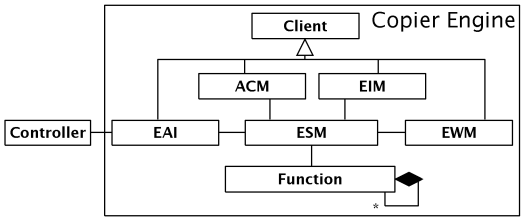


Figure 2. Architecture for copier engines

in Section 1. For this Océ relies on a model-driven approach based on UML [11]. Architects specify use case realisations using UML sequence diagrams. Based on these sequence diagrams, for every component a UML statechart diagram is created. Using special tooling[1], the source code for the engine components (e.g., ESM) is largely generated based on those statechart diagrams. For Océ's developers these statechart diagrams actually *are* the implementation.

One of the reasons for introducing a (automated) model-driven development approach was to overcome consistency problems with respect to state machine models and source code [11]. By automatically generating source code from state machines this problem is effectively moved 'upwards' to the consistency between scenarios and state machines.

For ESM, each use case addresses a specific engine state transition. A use case is accompanied by a UML sequence diagram. As an example, consider the diagram in Figure 6(a). It depicts the interaction that occurs when a copier engine is requested to go to standby, while it is running. At Océ these sequence diagrams are purely used for communication purposes, rather than input for automatic processing (e.g., model transformations, or code generation). Because of this, they are not always complete and precise. Furthermore, proprietary (non-UML) constructs are used. As an example, in these sequence diagrams the lifeline of the ESM component is decorated with the name of its (high-level) state at that point of the interaction.

To ensure successful evolution and maintenance of the reference architecture and the components it defines, a means to assess the consistency of the involved behavioural specifications is essential. It is this challenge we address in this paper.

[1]IBM Rational Rose RealTime - `http://www.ibm.com/software/awdtools/developer/technical/`

3. Related Work

Several formal approaches have been proposed that address problems similar to ours. Lam and Padget [12] translate UML statecharts into π-calculus to determine behavioural equivalence using bisimulation. Schäfer et al. [13] presents a tool that uses model checking to verify state machines against collaboration diagrams. The use of such tools and approaches requires complete, precise and integrated interaction- and state-based behavioural models. This implies, for instance, that sending and reception of messages in scenarios are explicitly linked to events and effects in state machines. In our case, for the sequence diagrams, this is problematic. They are created early in the development process and not intended to be complete or precise.

To take this into account, we generate a state machine from a set of input scenarios, that, subsequently, is compared to the state machine that was created by the developers.

Many approaches have been defined for synthesis of state-based models from scenario-based models. Amyot and Eberlein [8], and Liang et al. [14] both evaluate over twenty of them. Evaluation criteria include languages, means to define scenario relationships and state model type. Our industrial case gives us the requirements with respect to these criteria for a synthesis approach.

Instead of using a more powerful scenario language such as live sequence charts [15], we limit ourselves to UML sequence diagrams augmented with decorations, as dictated by our industrial case study. The decorations with state information can be interpreted as conditions from which inter-scenario relationships can be derived. Finally, with respect to state model type, we consider approaches that result in state models for individual components (instead of global state models). Considering Liang et al. [14] one approach best meets these requirements [10].

Whittle and Schumann [10] present an algorithm to map UML sequence diagrams to UML statecharts. In this mapping the messages in a scenario are first annotated with pre- and postconditions on state variables, referred to as a domain theory. The mapping is based on the assumption that a message only affects a state variable if its pre- or postcondition explicitly specifies it does; the domain theory does not need to be complete. Thus, this so-called frame axiom , together with the pre- and postconditions, results in a pair of state vectors for each message (before and after). For every scenario it is checked whether it (the message ordering) is consistent with the domain theory. If not, either one can be reconsidered. Then, for each scenario a 'flat' state machine is generated for every component. Messages towards a component result in an event that triggers a transition; messages directed away from a component result in an action that is executed upon a transition. Loops are identified by detecting states that have unifiable state vectors. Two states vectors are unifiable if they do not specify different values for the same state variable. Subsequently, the 'flat' state machines generated for a component from different scenarios are merged by merging similar states. Two state are similar if their state vector is identical and they have at least one incoming transition with the same label. Hierarchy is added to the resulting statecharts by a user provided partitioning and (partial) ordering of the state variables.

Most work in this area focusses on the synthesis algorithm, whereas the integration in industrial practice remains implicit. In fact, many of the approaches are not supported by a tool or validated in industrial practice. Their application in practice only becomes realistic when they integrate with existing tools and standards used in industry. Therefore, we focus in this paper on UML sequence diagrams as a notation for scenarios, and UML state machines.

4. Model-Driven Consistency Checking

In this section we outline our approach for consistency checking of behavioural specifications, but, first, we introduce the technologies that enable our model-driven approach and the underlying structure of the involved behavioural models.

4.1. Enabling Technologies

Our approach takes advantage of the standards that are widely used in industry, such as UML and XMI (XML Metadata Interchange), enabling easy integration with the tools used in industrial practice. XMI provides a means to serialise UML models to be manipulated, for instance, using XSLT (Extensible Stylesheet Language Transformations). However, the XMI format is very verbose, making it a tedious and error prone task to develop such transformations [16].

OMG's Model Driven Architecture (MDA) offers, among others, a solution to this problem. MDA is OMG's incarnation of model-driven engineering (MDE). With MDE, software development largely consists of a series of model transformations mapping a source to a target model. Essential to MDE are models, their associated metamodels, and model transformations. In the case of MDA, metamodels are defined using the MetaObject Facility (MOF). The UML metamodel is only one example of such metamodels. Finally, model transformation languages are used to define transformations.

We used the Atlas Transformation Language (ATL) [17] to specify and implement the mapping between scenarios and state machines. ATL is used to develop model transformations that are executed by a transformation engine. In ATL, transformations are defined in transformation modules that consist of transformation rules and helper operations. The transformation rules match model elements in a source model and create elements in a target model. To this end the rules define constraints on metamodel elements in a syntax similar to that of the Object Constraint Language (OCL). A helper is defined in the context of a metamodel element, to which it effectively adds a feature. Helpers can be used in rules, and optionally take parameters.

The ATL transformation engine can be used with XMI serialisations of models and metamodels defined using the MOF. For the sequence diagrams and state machines in this paper we used the MOF-UML metamodel available from the OMG [18]. To create the associated models, we use a UML modelling tool supporting XMI export.

Once the source model and metamodel, target metamodel, and transformation module are defined and located, the ATL transformation engine generates the target model in its serialised form, which, in turn, can be imported in a UML modelling tool for visualisation, or serve as source model for another model transformation.

4.2. Behavioural Modelling

For the creation of interaction-based and state-based behavioural models we use UML sequence and statechart diagrams. The underlying structure of these diagrams is described by the Collaborations and State Machines subpackages of the UML metamodel. Because our transformation rules are defined on the metamodel level, we introduce them briefly. Although we discuss only simplified versions of these packages, the implementation of our technique and our case study are based on the complete UML metamodel (version 1.4 [18]).

In general the UML specification [18] allows every model element to be associated with a set of constraints. We use this to add pre- and postcondition to Messages and state invariants to states. To distinguish between preconditions, postconditions, and other constraints that might be used in the model we use stereotypes.

Source: Collaborations The Collaboration package and some other UML elements are depicted in Figure 3. In the context of a Collaboration the communication patterns performed by Objects are represented by a set of Messages that is partially ordered by the predecessor relation. For each message sender and receiver Objects are

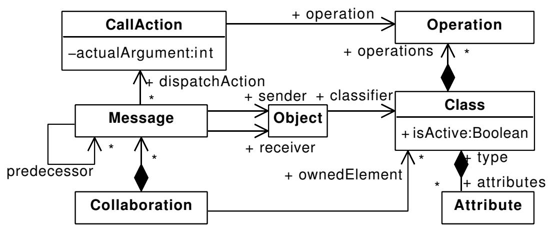


Figure 3. Collaborations (simplified)

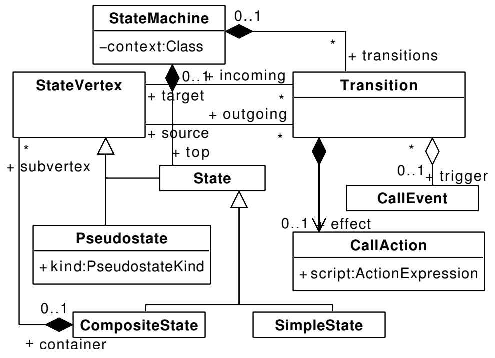


Figure 4. State machines

specified. As such, a Collaboration can be seen as the *specification* of one or more scenarios. The cause of a Message is a CallAction (dispatchAction) that is associated with an Operation. In turn, this Operation is part of the Class that is the classifier of the Object that receives the Message. Finally, a Class optionally contains Attributes that have a type.

Target: State Machines Using the (target) metamodel in Figure 4, UML state machines can be constructed that model behaviour as a traversal of a graph of state nodes interconnected by transition arcs.

A state node, or StateVertex, is the target or source of any number of Transitions and can be of different types. A State represents a situation in which some invariants (over state variables) hold. The metamodel defines the following types of States. A CompositeState contains (owns) a number of sub-states (subvertex). A SimpleState is a State without any sub-states.

Next to state nodes that describe a distinct situation, the metamodel also offers a type of StateVertex to models transient nodes: Pseudostate. Only one Pseudostate type (PseudostateKind) is relevant for the state models in this paper: initial Pseudostate. An initial Pseudostates is the default node of a CompositeState. It only has one outgoing Transition leading to the default State of a CompositeState.

Nodes in a state machine are connected by Transitions that model the transition from one State (source) to another (target). A Transition is fired by a CallEvent (trigger). The effect of a Transition specifies an CallAction to be executed upon its firing. Finally, a StateMachine is defined in the context of a Class and consists of a set of Transitions and one top State that is a CompositeState.

4.3. Consistency Checking Approach

As said, the set of scenarios is not expected to be complete or precise. For instance, when comparing, the set of scenarios and the state machines created by the developers it is unclear whether a scenario specifies universal or existential behaviour [15]. However, if we are to generate a state machine for a set of scenarios we have to take a position with respect to the meaning of those scenarios. The generation of scenarios is based on the approach in Whittle and Schumann [10]. For this, we interpret Océ's scenarios in principle as universal. This means that if the start condition of a scenario is satisfied the system behaves exactly as specified by that scenario. We consider the start condition of a scenario to be the first condition specified as decoration and occurrence of the first message. As such, the scenario in Figure 6(a) specifies exactly what happens when ESM receives the message m_SetUnit(standby) while it is in state running. However, when during execution of a scenario the start condition of another scenario is satisfied, execution continues according to that scenario. For instance, in the case of Figure 6(a), while ESM is stopping, execution could continue according to the scenario that performs the request of ESM going back to running while it was stopping.

In our approach we use model transformations for the generation of a state machine from a set of scenarios. The specification of those transformations is discussed in Section 5. To include all required information, the source model has to comply to a set of modelling conventions. When considering an arbitrary industrial case (e.g., Océ's reference architecture), the models used typically do not comply to those conventions. Therefore, we first require models to be normalised. This is discussed in Section 6.

Finally, the generated state machine is compared to the state machine that was already developed based on the same set of scenarios, the implementation state machine. Because the sequence diagrams are created early on in the development process, it is not expected that they are exactly covered by the state machines. Therefore, mismatches are expected between the generated and implementation state machine with respect to transition labels and order. This makes automating the comparison step particularly difficult. For now we manually compare the generated and implementation state machine and mainly focus on inconsistencies with respect to top-level states and transitions.

As such, we use three steps to check to consistency of behavioural specifications: normalise, transform, and compare. In the current approach only the transformation is automatic. Furthermore, the normalisation step is context-specific as it depends on the type of input models.

5. Generating State Machines

Given the source and target metamodels discussed in the previous section, we now describe how to instantiate source models, as well as the mapping between source and target models, expressed as ATL model transformations. We published all (executable) ATL transformations that we implemented, as well as (normalised) source and target (meta)models for the ATM example of Whittle and Schumann [10] in the ATL Transformations Zoo [19].

5.1. Instantiating a Source Model

Our approach based on model transformations and UML requires that all necessary information is encoded in a UML model. Whittle and Schumann [10] requires the following information for its mapping: scenarios, a domain theory, a set of state variables, and an ordered partition of that set.

The set of scenarios is specified as sequence diagrams. The types of the interacting Objects (components) are specified in a class model. The Class that corresponds to the component of interest is marked active. All Operations involved in the relevant scenarios are also specified. The pre- and postconditions of a domain theory are applied to these Operations as stereotyped Constraints. These Constraints have the form `state variable = value`. We currently do not allow pre- and postconditions in the domain theory that refer to formal parameters, as this would require interpretation of these conditions. If necessary, such constraints can be added directly to the Messages that specify an actual parameter in the sequence diagrams.

The active Class contains an Attribute for each state variable. The partition of state variables used for introducing hierarchy is encoded by setting the visibility of all state variables included in the partition to public and the others to private. Finally, the order of the state variable Attributes on the Class represents the prioritisation of state variables (the top one having the highest priority).

5.2. Model Transformations

Our transformations generate a state machine for the component that is represented by the active Class in the source model. A scenario specifies one particular path through the state machine for that component, on which it proceeds to the next state upon each communication. We refer to the state machine that only describes that path as a 'flat' state machine.

We tailored the approach in Whittle and Schumann [10] (see Sec. 3) to account for the type of input in the Océ case, our model-driven strategy, and for our goal: consistency checking. For this reason we introduce less abstractions. This makes detecting and resolving inconsistencies more convenient. Our mapping consists of four separate steps: 1) apply domain theory, 2) generate flat state machines, 3) merge flat state machines, and 4) introduce hierarchy to merged state machine.

We formalised our mapping from scenarios to state machines as four ATL model transformations that correspond to the four steps of our mapping. Every consecutive transformation uses the target model of the previous transformation as its source model.

Together, these transformations are specified in less than 700 lines of ATL code. Before these transformations can be applied to the Océ case, a normalisation step is required, which is discussed in Section 6.

Apply Domain Theory This step is specific to our approach. Unlike Whittle and Schumann [10], but in accordance with the UML, we distinguish between pre- and postconditions on the Operations of a Class and on the CallActions associated with Messages in a sequence diagram. This has two advantages. First, it allows for simple pre- and postconditions to be specified only once (i.e., on the Operations of a Class). Second, it circumvents the need to evaluate conditions that refer to formal parameters of an Operation.

When we apply the domain theory to a set of scenarios, we simply attach the pre- and postconditions on the Operations of a Class to corresponding Messages to or from instances of that Class.

The ATL specification of this mapping is straightforward. The Constraints on an Operation are copied to Messages, via their associated CallAction. Listing 1 specifies a rule that matches all CallActions. For each it generates a CallAction, `ca_out`, in the target model and initialises its constraint feature with the constraints applied to the Operation associated with the matching CallAction. Note that the constraints are added to the constraints already applied to the matched CallAction (using the `union` operation).

The result is a set of sequence diagrams in which

```
rule ConstrainedCallAction {
 from ca_in:UML!CallAction
 to ca_out:UML!CallAction(
   operation <- ca_in.operation,
   constraint <- ca_in.operation.constraint->union(
    ca_in.constraint))
}
```

Listing 1. Applying constraints to CallActions

Constraints are applied to Messages based on the pre- and postconditions of a domain theory on Operations. See Figure 6(b) for an example.

Sequence Diagrams → Flat State Machines The next step of our approach is to generate a flat state machine for every scenario in which the component of interest plays a role. In this step we map every communication to a Transition and a target State. The source State of this transition is the target State corresponding to the previous communication of the component in the scenario. As in the approach in Whittle and Schumann [10]; if the involved communication was the receipt of a Message, we say the Transition was triggered by that Message. If the involved communication was the sending of a Message, we say the effect of the Transition was sending that Message.

Based on the pre- and postconditions applied to the Messages in the scenarios by the previous step, we calculate the state vector for each State. For this we 'propagate' pre- and postconditions through the sequence diagram by application of the frame axiom. The result is a set of flat StateMachines, in which state vectors are applied to States as a set of Constraints over state variables.

As an example, the `EffectTransition` rule in Listing 2 matches all Messages in the source model sent by the component of interest. The target pattern specifies that for each such Message (`m`) among others, a Transition (`t_effect`) and a SimpleState (`trgt`) are created in the target model. The effect and target features of the Transition element are simply initialised to the CallAction (`ca`) and SimpleState created in the same rule. The source of the Transition is initialised to the target of the Transition that correspond to the previous Message (not shown).

The constraint feature of the generated SimpleState element is initialised to the set of constraints (state invariants) that hold after the Message that matched the rule. This is determined by the `stateVector` helper. For this it applies the frame axiom (specified in the `frame` helper) subsequently to the postconditions of the current Message (`'posts'`), the preconditions of the current Message (`pres`), and the state vector after the previous Message (`stateVectorPrev`). As such conditions propagate in

```
rule EffectTransition {
 from m:UML!Message (m.sender.isActive)
 to t_effect: UML!Transition(
   effect <- ca,
   target <- trgt,
   source <- ... ),
  ae:UML!ActionExpression ( ... ),
  ca:UML!CallAction ( ... ),
  trgt:UML!SimpleState (
   name <- ae.body+'_sent',
   constraint <- m.stateVector)
}
helper context UML!Message def: stateVector : Set(UML
 !Constraint) =
 let stateVectorPrev:Set(UML!Constraint) = ... in
 let pres:Set(UML!Constraint) = ... in
 let posts:Set(UML!Constraint) = ... in
 let sv:Set(UML!Constraint) =
thisModule.frame(stateVectorPrev,thisModule.frame(
 pres,posts)) in
   if thisModule.unifiable(stateVectorPrev,pres) then
     sv
   else
     sv.debug('INCONSISTENCY DETECTED!')
   endif
;
helper def: frame(frame:Set(UML!Constraint), framed:
 Set(UML!Constraint)): Set(UML!Constraint) =
  frame->iterate(c; cs:Set(UML!Constraint)=framed |
   if cs->exists(e|e.stateVariable=c.stateVariable)
    then
     cs
   else
     cs->including(c)
   endif)
;
```

Listing 2. Message →effect Transition

'forward' direction (i.e., downwards in a sequence diagram).

Additionally the `stateVector` helper notifies the user if an inconsistency is detected between the state vector after the previous Message and the preconditions for the current Message (these sets of Constraints should be unifiable).

The `frame` helper simply iterates over the Constraints in the `frame` argument and adds every constraint involving a state variable that is not referred to in `framed` to that set.

Unlike Whittle and Schumann [10] we do not apply unification of state vectors at this stage. The declarative style of our ATL specifications results in an infinite recursion: to complete a state vector we need to know whether it can be unified with other state vectors. To determine this we have to consider state vectors in 'forward' as well as in 'backward' direction. However, the state vectors in 'forward' direction, in turn, consider state vectors in 'backward' direction because of the frame axiom strategy.

Application of this step yields a set of flat state machines for a component. As an example, consider Figure 5. It depicts the flat state machine corresponding to the sequence diagram in Figure 6(b). Note that the ex-

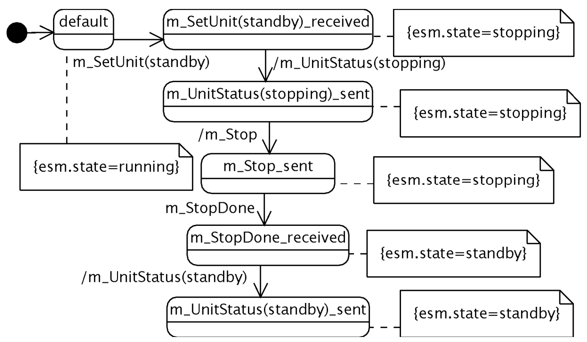


Figure 5. Flat state machine

ample only involves a single state variable and that the names of the States are derived from the particular Message that was sent or received by the component.

Merging Flat State Machines In this step we merge the flat state machines. We merge every set of states with unifiable state vectors and identical incoming transition (in terms of effect or trigger) into a single state.

Merging of states is done by the rule and helpers in Listing 3. The rule matches all states selected by the `mergedStates` helper that iteratively selects one SimpleState from every group of equal SimpleStates in the source model. A call to the `mergeable` helper results in true when 1) the receiving StateVertex and the parameter StateVertex (`s`) are unifiable, and 2) have the same name (i.e., the incoming transitions had the same trigger or effect). The `unifiable` helper evaluates to true for two sets of Constraints that do not specify different values for the same state variable, meaning that the constraint that refers to a particular state variable that is also referred to in the other set, is actually included in that set.

Transitions are matched by another rule (not shown). To discard redundant Transitions, it only matches one Transition of the Transitions between any two sets of SimpleStates that are merged.

Introducing Hierarchy As suggested by Whittle and Schumann [10] we use an ordered partition of the set of state variables to add hierarchy by means of CompositeStates. The problem here, is that there is not always a matching source model element to create a CompositeState for. Therefore, we use a *called* rule (`CompositeState`). A called rule is an imperative rule that is not matched by a source model element, but is explicitly called and can have parameters. This rule creates a CompositeState for a given set of Constraints (`cseq`). These Constraints (i.e., state invariants) are determined by the `compositeStateConstraintSetsAt` helper that takes a set of Constraints that represents the current CompositeState and determines the sets of Constraints that correspond to the CompositeStates at that level. For each of those sets a CompositeState is created. This called rule is used to initialise the `subvertex` feature in the rule that matches the top CompositeState of the merged StateMachine, as well as (recursively) in the `CompositeState` rule itself. The `do` clause in the `CompositeState` rule returns the created CompositeState.

```
rule MergedSimpleState {
  from s_in:UML!SimpleState (
    thisModule.mergedStates->includes(s_in))
  to s_out:UML!SimpleState(
    name<-s_in.name,
    constraint <- s_in.constraint)
}
helper def: mergedStates: Set(UML!StateVertex) =
  thisModule.allSimpleStates->union(thisModule.
   allPseudostates)
  ->iterate(s; mss:Set(UML!StateVertex)=Set{} |
    if mss->exists(e|(e.mergeable(s)) then
      mss
    else
      mss->including(s)
    endif)
;
helper context UML!StateVertex def: mergeable(s:UML!
 StateVertex): Boolean =
  thisModule.unifiable(self.constraint,s.constraint)
   and self.name=s.name
;
helper def: unifiable(cseq1:Sequence(UML!Constraint),
 cseq2:Sequence(UML!Constraint)): Boolean =
  cseq1->includesAll(cseq2->select(c|cseq1->collect(e|
   e.stateVariable)->includes(c.stateVariable)))
;
```

Listing 3. Merging SimpleStates

```
rule TopCompositeState {
  from cs_in:UML!CompositeState
  using {
    sm:UML!StateMachine=thisModule.allStateMachines->
     select(sm|sm.top=cs_in);
  }
  to cs_out:UML!CompositeState (
    name <- cs_in.name,
    subvertex <- sm.simpleStateStatesAt(Set{})
    ->union(sm.compositeStateConstraintSetsAt(Set{})
      ->collect(cs|thisModule.CompositeState(sm,cs))))
}
rule CompositeState (sm:UML!StateMachine, cseq:Set(
 UML!Constraint)) {
  to cs:UML!CompositeState(
    subvertex <- sm.simpleStateStatesAt(cseq)->union(
     sm.compositeStateConstraintSeqsAt(cseq)->collect(
     cs|thisModule.CompositeState(sm,cs))))
  do{cs;}
}
```

Listing 4. Adding hierarchy to state machine

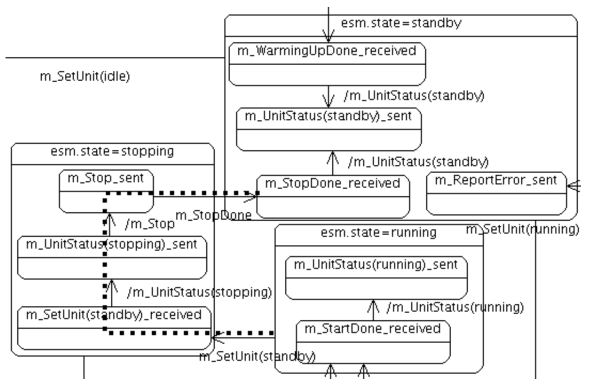


Figure 7. Merged state model of ESM (fragment)

6. Normalising the Source Model

In the case of Océ, neither a domain theory, nor a set of state variables were available. To overcome this, we normalise Océ's sequence diagrams. In particular, we interpret the decorations on object lifelines as pre- and postconditions on a single state variable: `state`. The message preceding a state decoration apparently resulted in the component moving to the indicated state. Hence, we (manually) attach a corresponding postcondition (e.g., `esm.state=starting`). A message succeeding a state decoration apparently requires the component to be in the indicated state. Hence, we attach a corresponding precondition. As an example, consider Figure 6. Finally, we added a (public) attribute, `state`, to the class corresponding to the ESM component.

7. Results

A fragment of the result of application of the transformation step to Océ's ESM component, is depicted in Figure 7. The dashed line indicates the path through the state machine that is traversed when ESM is requested to go to standby while it is running. This path corresponds to the scenario depicted in Figure 6.

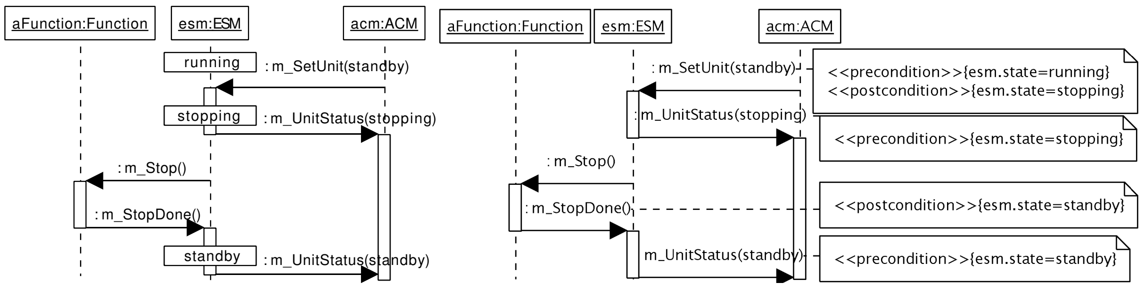
We compared this *derived* state machine with the *implementation* state machine, from which Océ generates code. There are many inconsistencies with respect to low-level states and transitions. In the implementation state machine low-level states are not only decomposed further, the sequence of states and transitions is also different in many cases. This is not surprising considering the fact that the sequence diagrams of the source model from which we derived a state machine, constitute the first behavioural model that is created for the ESM component, while, in the implementation state machine, low-level transitions and states often correspond to a single method call in the generated code. If we restrict the comparison step to the top-level states, however, the implementation state machine largely conforms to the derived state machine. Although we cannot show the implementation state machine, we were able to make several other interesting observations:

- Several transitions between top-level composite states are missing in the derived state machine. This indicates not all scenarios have been specified in a sequence diagram.
- Some top-level composite states in the derived state machine were modelled as low-level (sub) composite states in the implementation state machine. This merely indicates changes to the decomposition of states, and does not necessarily result in different behaviour.
- In the derived state machine, sometimes extra paths exists between two composite states. This indicates specific sequences of events and actions that occur in different scenarios are not specified consistently. This was the case, for instance, when two versions of a scenario existed: one for normal behaviour, and one for exceptional behaviour. For two such versions the first interactions should typically be identical (until some exception occurs), but in practice this was not the case.
- The derived state machine contains a number of unconditional transitions that form a loop, resulting in non-deterministic behaviour. This had the same cause as the previous observation.

As a response to these observations Océ could decide to add missing use cases and scenarios, and to refactor alternative sequence diagrams to remove inconsistencies in event and action sequences. Here, care must be taken, as such modifications affect the state machines of other components that play a role in the involved scenarios as well. On the other hand, if such steps are not taken and behavioural inconsistencies are only removed in the implementation state machine, other development groups, responsible for other components, might do so differently, resulting in integration and maintainability problems.

Although, the normalised source model in the Océ case only contains a single state variable, we also applied our transformation step to the ATM example in Whittle and Schumann [10][1]. This example involves three state variables. By application of our approach (in both cases) we detected several inconsistencies.

[1]Images of the (normalised) source model, as well as all (intermediate) target models for the ATM example can be downloaded from the ATL Transformations Zoo [19]



(a) Sequence diagram with decorated lifeline (b) Normalised sequence diagram

Figure 6. Example scenario: request a copier engine to go to standby while it is running

8. Discussion

Generalisability of the approach *To a large extent our approach is generic.*

We applied our approach successfully to both Whittle and Schumann [10]'s ATM example and Océ's reference architecture. Our approach is generic with respect to input models that comply to the model conventions as outlined in Section 5.1. As such, we require a (manual) normalisation step that is context specific; it depends on the modelling conventions in use at a particular company.

Our modelling conventions are most restrictive with respect to the type of pre- and postconditions used in the domain theory. As we do not evaluate these conditions, we require them to be of the form `stateVariable=value`. In the case the conditions for an Operation refer to a formal parameter. Our approach can still be applied if the Messages associated with that Operation in the sequence diagrams specify a corresponding actual parameter. Then, we (manually) apply the condition directly to the Message in the sequence diagram and substitute the formal parameter for the actual parameter. More complicated conditions requires real interpretation of OCL expressions.

Of course, pre- and postconditions have to be available for our approach to produce more than only flat state machines. In the case of Océ's, we derived pre- and postconditions from decorations in the sequence diagrams. In general, pre- and postconditions are not always obvious from design documentation. In such situations these might have to be derived indirectly from documentation or reverse engineered from source code.

The introduction of pre- and postconditions effectively is a normalisation to the UML standard used by Océ and our tools (version 1.4 [18]). For the latest UML (version 2.0) this is not necessary, as such lifeline decorations became part of the specification (the corresponding metamodel element is called StateInvariant). To support this, only minor modifications to our ATL transformations are required.

Scalability of the approach *Our approach constitutes a first step towards fully automated consistency checking.*

In the Océ case, the source model for the transformation step includes 10 sequence diagrams that specify 62 messages. The resulting integrated, hierarchical state machine, of which a fragment was depicted in Figure 7 contains 23 transitions between 14 composite states containing in total 47 simple states.

Our approach is a first step to fully automated consistency checking of behavioural specifications. For now, we rely on manual inspection of the resulting state machine for actual evaluation of the consistency. As such, the scalability is currently not limited by the transformation steps (in the Océ case they each take less than seconds), but by the comparison step. For cases were the number of states is limited and developers have knowledge on the system, this is a feasible approach. For ESM, which is a medium-sized component (approximately 10 KLOC), this turned out not to be a problem.

Automatic consistency checking could be done by relying on naming. An example of such an approach is discussed in Van Dijk et al. [16]. It checks the consistency of the underlying XMI representations of UML models. In general this problem is equivalent to graph matching. Also for automatic approaches, however, the generation of a state machine from a set of scenarios, as discussed in this paper, is likely to be a first step.

Applicability of the approach *Our approach can be applied to iteratively develop behavioural specifications.*

We generated a state machine with the purpose of checking the consistency of different behavioural specifications. However, our approach might have other types of applications as well. A generated state machine could also be used for other types of analyses, such as model checking or performance analysis.

Next to analysis purposes, our approach is particularly also interesting for forward engineering, especially in the context of model-driven development approaches as in the case of Océ. Using our transformations based on UML, developers can easily generate different views on the behaviour of a software system or component. Furthermore, the generation not only provides insight in the consistency of the sequence diagrams with respect to each other, it also provides developers with a first candidate state machine that can be refined. As such, our technique can be applied iteratively to develop complete behavioural specifications of components: (1) specify the interactions of an initial set of use cases as scenarios, (2) generate a state machine, (3) refactor scenarios to remove inconsistencies in event and action sequences, and add missing scenarios, (4) goto step 2.

The main reason to choose for a model-driven approach based on UML for our consistency check, was the integration with Océ's development process. It circumvents the need to extract information from the MDA domain to another domain, e.g, the grammarware, or XML domain. Unfortunately, despite the availability of standards, currently available tools for (meta)modelling and transformations do not integrate well, hampering actual integration of our approach in practice. For a large part this is due to the abundance of possible combinations of XMI, UML, and MOF versions, as well as vendor specific implementation of those standards. Other problems occur due to different capabilities of modelling tools. As an example, we used Poseidon for UML to create source models because its metamodel is available from the developer's website. However, the UML models we generate do not contain layout information. Unfortunately, Poseidon is not capable of displaying UML models that do not contain layout information. As a consequence we had to use another tool for visualisation. From a large set of tools we tried, only Borland's Together is capable of generating a layout for a UML model. However, the XMI representations used by this tool are not compatible with those generated by the ATL engine. As a workaround we developed a minimal XSLT transformation that maps the XMI 'flavour' generated by the ATL engine to that of Together. An alternative is to generate the layout information required by Posedion using a model transformation.

UML vs. MOF *The use of* UML *in a limited domain makes transformation definitions unnecessary complex*

The genericity and resulting complexity of the UML metamodel result in, sometimes, inconvenient navigation through source and target models to select a certain element. Also, often relations are defined as $n : n$ while in a specific case $1 : 1$ would suffice. The result is that sets have to be converted to sequences of which the first element has to be selected. This is required very frequently, resulting in unnecessary complex ATL-code.

In cases, where only limited parts of the UML metamodel are used, an alternative could be considered. Instead of using the UML metamodel, custom MOF-based metamodels could be used, for instance, for scenarios and state machines. These metamodels could be much simpler, resulting in simpler transformation definitions.

9. Conclusions

In this paper we demonstrated the use of model transformations to check the consistency of behavioural specifications. For this we presented an approach that consist of normalisation, transformation, and comparison steps. We consider the following to be the main contributions of this paper:

- A specification of the mapping between scenarios and state machines using model transformations that is made available via the ATL Transformations Zoo [19]. An advantage of such a specification is that it can be executed by the ATL transformation engine. Furthermore, it is completely based on UML, allowing easy integration in industrial practice.
- Modelling conventions for encoding the information required for the transformation step in a single UML model. Additionally, as an example, we discussed the required normalisation step for Océ's reference architecture.
- Validation of the proposed approach by application to an industrial system, resulting in the identification of a number of inconsistencies in its behavioural specifications.

Finally, the proposed approach could be applied for other purposes than consistency checking as well, such as forward engineering and early behavioural analysis based on the generated state machine.

Currently we are extending our work with additional case studies. Furthermore, we investigate the possibilities to do consistency checking automatically. Again, by the use of MDA model transformation technologies.

Acknowledgement Part of the research described in this paper was sponsored by NWO via the Jacquard Reconstructor project. Furthermore we would like to thank Océ, and in particular Lou Somers for providing the case study.

References

[1] M. M. Lehman and L. A. Belady, eds. *Program evolution: processes of software change*. Academic Press, 1985.

[2] A.P. Bröhl and W. Dröschel. *Das V-Modell. Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenbourg-Verlag, München, 2nd edition, 1995.

[3] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[4] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Addison-Wesley, 1998.

[5] Phillipe Kruchten. *The Rational Unified Process*. Addison-Wesley, 1998.

[6] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[8] Daniel Amyot and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1), September 2003.

[9] Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Engineering*, 31(12):999–1014, December 2005.

[10] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proc. 22nd Int'l Conf. Software Engineering (ICSE 2000)*, pages 314–323. IEEE CS, 2000.

[11] L. A. J. Dohmen and L. J Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In *Proc. PROFES 2002*, volume 2559/2003 of *LNCS*, pages 475–484. Springer-Verlag, 2003.

[12] Vitus S.W. Lam and Julian Padget. Analyzing equivalences of uml statechart diagrams by structural congruence and open bisimulations. In *Proc. 2003 IEEE Symposia on Human Centric Computing Languages and Environments (HCC 2003)*, pages 137–144. IEEE CS, October 2003.

[13] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. In *Proc. Workshop on Software Model Checking*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 357–369. Elsevier, 2001.

[14] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proc. 5th Int'l Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006)*, pages 5–11. ACM, 2006.

[15] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[16] Hylke W. van Dijk, Bas Graaf, and Rob Boerman. On the systematic conformance check of software artefacts. In *Proc. 2nd European Workshop on Software Architecture (EWSA 2005)*. Springer-Verlag, June 2005.

[17] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proc. Model Transformations in Practice Workshop at MoDELS2005*, 2005.

[18] OMG. OMG Unified Modeling Language Specification, Version 1.4. `http://www.uml.org`, 2001.

[19] ATL Transformations Zoo. `http://www.eclipse.org/gmt/atl/atlTransformations/#UMLSD2STMD`.