Migrating Supervisory Control Architectures Using Model Transformations

Bas Graaf
TU Delft
The Netherlands
b.s.graaf@ewi.tudelft.nl

Sven Weber
ASML and TU/e
The Netherlands
sven.weber@asml.com

Arie van Deursen
CWI and TU Delft
The Netherlands
arie.van.deursen@cwi.nl

Abstract

This paper describes an approach for the migration of supervisory machine control architectures. This migration, from a paradigm based on finite-state machines to a paradigm based on task-resource systems, is described in terms of model transformations. We propose a generic migration approach that involves normalising a legacy architecture that, in turn, is transformed. Based on the architecture of a controller of a complex manufacturing machine, a wafer scanner developed by ASML, we define a number of concerns and corresponding architectural transformation rules.

1. Introduction

As software systems evolve they tend to become increasingly complex [1]. Furthermore, the architecture documentation and its corresponding implementation tend to follow asynchronous evolutionary paths. Consequently, the conformance between the architecture specification and software implementation decreases as a software system evolves [2].

In practice, increased complexity and loss of architecture conformance make a system more difficult to change [3], and result in an increase in development (and maintenance) effort and cost. The involved effort can be reduced by, for example, the separation of concerns, the introduction of product-line architectures, model-based development and automatic code generation. In practice, adopting such techniques will require architectural changes.

Ideally, one would like to make these changes reproducible by automatically transforming one architecture into another. When migrating towards a product line, such a transformation could be applied repeatedly to migrate different product versions into product-line members. In this paper we consider the migration of supervisory machine control (SMC) architectures towards such a product-line approach.

In an advanced manufacturing machine, SMC [4, 5] is responsible for the coordination of the high-level machine behaviour. This requires, amongst others, interpretation of manufacturing requests and prioritisation, synchronisation, scheduling, and exploitation of concurrency with respect to the resulting manufacturing activities [6, 7, 8, 9]. For advanced manufacturing machines, these control systems have an indicative order of magnitude of 10 SMC components, each encompassing 10^5 lines of code.

The SMC of a wafer scanner, developed by ASML, motivated this paper and serves as a running example to illustrate the migration of a legacy architecture, based on finite-state machines (FSM's), to a new architecture that is based on task-resource systems (TRS's). This migration is spurred by the fact that a TRS-based SMC architecture, as opposed to an FSM-based one, is declarative, supports run-time dependent decisions and thus provides increased flexibility and maintainability.

We consider the start and end point of this migration as different architecture views, that we refer to as the source and target view respectively. This is similar to the approach for architecture reconstruction as described by Van Deursen et al. [10]. Here, an architecture view is associated with a viewpoint [11], that, amongst others, specifies a metamodel for the primary presentation [12] of that view. In this paper we focus on the models in this primary presentation.

In order to define a reproducible mapping, we define practical transformation rules in terms of patterns associated with the source and target metamodels. These transformation rules are practical in the sense that they are based on an actual migration as was performed by an expert. From this migration, we have extracted generic, concern-based transformation rules. Due to practical reasons, which are mainly associated with the informal use of modelling languages in industry [13], we first normalise the legacy specification before the architecture transformation rules are applied via a model transformation language. The resulting migration approach, as depicted in Figure 1, is similar to the MDA framework [14] except for this normalisation step.

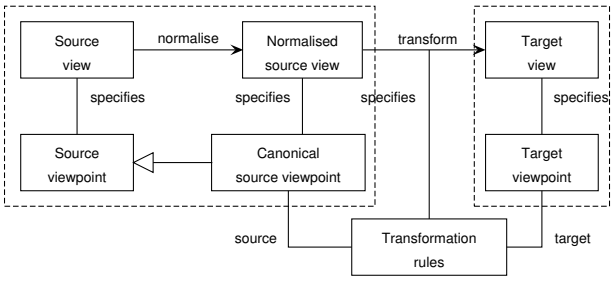


Figure 1: Generic two-phased migration approach

Although we focus on the migration of SMC architectures in particular, the principles as outlined in this paper can be applied to a much broader class of systems. Advanced manufacturing systems in general, require some form of supervision for the coordination and execution of manufacturing activities. We believe that the main contribution of this paper is a practical view on the migration of supervisory control systems in general.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3 we introduce SMC, specific concerns, and our running example. The elements necessary for the proposed migration are presented and illustrated using the running example, in Sections 4, 5 and 6. Section 7 reflects on the migration. The paper is concluded in Section 8.

2. Related work

Fahmy and Holt [15, 16] discuss several types of generic architecture transformations that can be viewed as graph transformations. In this paper we consider domain-specific transformations on architecture views that are more complex than typed graphs. Moreover, the transformations discussed in this paper are driven by a migration to a completely different SMC paradigm.

Czarnecki and Helsen [17] give a categorisation of model transformation approaches. Here, the type of the transformation rules constitutes one of the main characteristics. In this paper, we provide *pattern-based* transformation rules using a *graphical syntax*. These rules are organised *independently* and can be *selectively* applied resulting in a target model.

Bosch [18] uses architecture transformations in architecture design to realise non-functional qualities of a system. One of the identified transformation types is the application of an architectural style. A migration towards a TRS-based SMC architecture could constitute such a transformation. In our case, however, this transformation also results in a product-line architecture, for which we investigate the derivation of product instances.

Research related to the MDA and, more particularly, on the involved model transformations ([19, 20]), provides another source of related work. In accordance to the MDA vision, most work focusses on the transformation to platform-specific models. However, these platforms are mostly middleware solutions for information systems.

3. Supervisory Machine Control

In this section we first define SMC and place it in a machine control context. Next, we briefly discuss the main concerns that need to be addressed in the architecture migration. Finally, we introduce the motivating case and running example for this paper: the ASML wafer scanner.

3.1. Context and control problem

The machine control context is clarified in Figure 2. The transducers, or sensors and actuators, of a manufacturing machine are mounted on (sub)frames and are coupled through mechanical and electronic interfaces. The regulative controllers calculate actuator set-points and trajectories as well as induce regulated changes in the system under control. We do not consider this class of control systems.

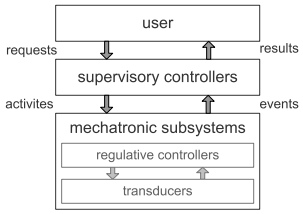


Figure 2: Machine control context

From a supervisory perspective, (sub)frames, transducers and associated regulative controllers form mechatronic subsystems on which activities can be executed that add value to products. The recipe- and customer-dependent routing of multi-product flows, with varying optimisation criteria, constitutes one of the key (supervisory) control issues. Moreover, advanced manufacturing machines must respond correctly and reproducibly to manufacturing requests, run-time events and results. Consequently, a supervisory controller is required to coordinate the execution of manufacturing activities and to ensure feasible machine behaviour [4, 6, 7, 8].

In practice, a high-level manufacturing request is translated into valid low-level machine behaviour using multiple, consecutive control-layers. This is supported by recursive application of the control context from Figure 2: activities of one level become manufacturing requests for the next level until the mechatronic subsystems are reached.

3.2. Typical concerns

For SMC of advanced manufacturing machines, a number of key (migration) concerns can be identified. Predominantly, multiple manufacturing activities - and sequences hereof - may fulfil a request and, in turn, multiple mechatronic subsystems may be available to perform a particular activity. That is, multiple alternatives exist that require the selection of a specific subset of both activities and mechatronic subsystems to fulfil a given manufacturing request. In other words, the SMC system has to make various control decisions with respect to *alternatives*.

The execution of an activity on a selected subsystem implies a specific physical state transition of that subsystem. The selected sequence of activities for a subsystem requires matching end states and begin states for consecutive state transitions. When these states do not match, an additional transition, a setup, has to be executed between consecutive activities. In SMC, these sequence-dependent *setups* are common.

Intuitively, controlled *usage* of mechatronic subsystems is another important concern. As such, the control system generally checks the availability of a subsystem that is required for a manufacturing activity. Once available, the subsystem should be effectively claimed for the given activity. When an activity has been (co)performed by claimed mechatronic subsystem(s), all should be unclaimed or released.

In order to take full advantage of installed capacity, *concurrent execution* of activities is done where possible. In practice, activities can be executed concurrently unless this is explicitly prohibited by precedence (sequence) relations. *Synchronous execution* is another common concern. Physical space is often limited resulting in multiple mechatronic subsystems that simultaneously operate within a confined space. This results in so-called hazardous areas in which subsystems can collide and state transitions must be induced synchronously to ensure safety.

3.3. Running example: a wafer scanner

In this paper we consider the ASML wafer scanner as a representative example of an advanced manufacturing machine. Wafer scanners are used in the semiconductor industry and perform the most critical step in the manufacturing process of integrated circuits (IC's). Figure 3 illustrates a scanner and its subsystems.

A neighbouring machine, the track (TR), performs preprocessing steps and delivers silicon wafers to the prealignment system (PA). Here, the wafer orientation and alignment are determined and adjusted. Next, the load robot (LR) transports the wafer to one of the two wafer stages (WS). Here, the wafer characteristics are measured.

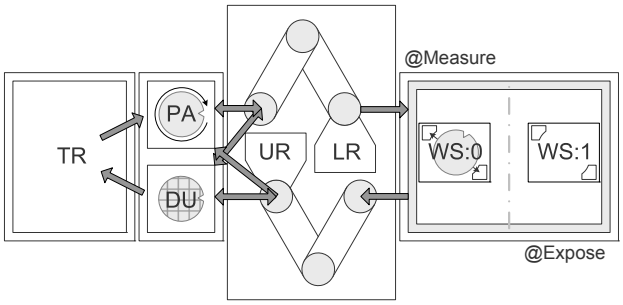


Figure 3: Simplified layout of a wafer scanner

After measurement, the wafer stages are swapped and the measured wafer is exposed. During exposure, a laser projects an image of the required IC pattern onto the wafer's surface through a demagnification lens. A wafer is exposed in a scanning fashion, similar to a photo-copier. Eventually, the wafer comes to hold hundreds of small copies (i.e. dies) of this pattern.

After exposure, the stages swap back and the unload robot (UR) transports the exposed wafer to the discharge unit (DU) where it is buffered. Next, the wafer is picked up by the track again to undergo various post-processing steps. Now, the wafer is ready for another exposure if needed: the process is re-entrant. With each passing, another layer is added to each die. Once the wafer has been fully processed and inspected, it is diced into individual dies that are packaged to form IC's such as microprocessors.

4. Source view: FSM-based SMC

The source view of the migration is based on finite state machines (FSM's). This approach is proposed by, e.g., Ramadge and Wonham [4]. Here, the set of possible machine behaviours is considered to form a language. A discrete supervisory FSM is synthesised that restricts this language by disabling a subset of events to enforce valid machine behaviour. This requires the behaviour in all possible states to be specified explicitly using (un)conditional state transitions with associated triggers (events) and effects (manufacturing activities). Hence, the machine behaviour for each type of manufacturing request, and combinations hereof, is specified design-time. Consequently, multiple FSM's are used per controller (typically one for each type of request).

In our case, the source view is specified using UML state machine diagrams. The relevant part of the UML metamodel is shown in Figure 4. As an example of how this metamodel is used in practice, consider the source view of the unload wafer request as depicted in Figure 5. In general, a single FSM-based SMC component has an indicative order of magnitude of 10^2 states and 10^3 transitions.

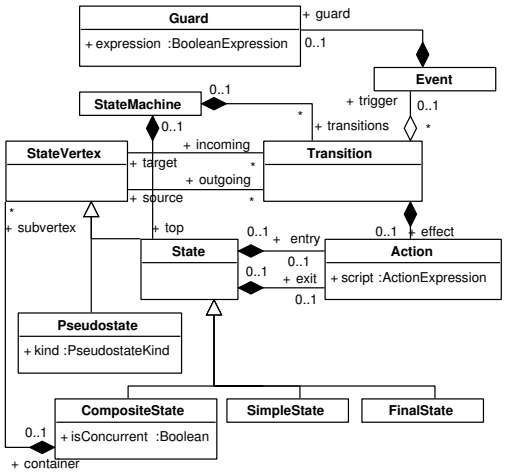


Figure 4: Source metamodel (excerpt from [21])

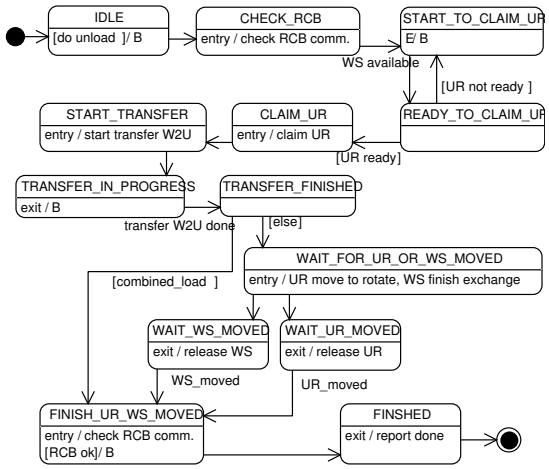


Figure 5: The unload wafer request

UML, as a generic modelling language, lacks constructs to support its application in the domain of SMC systems. Furthermore, appropriate profiles are seldomly used in practice. This makes that, when using 'plain' UML, various design idioms are available for handling SMC-specific concerns. For instance, guards (e.g., 'subsystem is available') can be modelled as events (e.g., 'subsystem becomes available') although these are fundamentally different concepts. Similarly, manufacturing activities can be specified as actions on state transitions or as actions in separate states.

In practice, specifications have a tendency to become inconsistent and ambiguous over time and are often incomplete. This phenomenon is further amplified by tool limitations. For instance, tools that support a specific UML version, do not necessarily support all of its constructs. When reconsidering Figure 5, these issues are illustrated by e.g. WS and UR appearing in two distinct resource usage idioms and the fact that, for the actual transfer, the alternative completion sequences are specified exhaustively.

To identify the idioms used for SMC concerns, legacy diagrams need to be manually inspected. This is a tedious and error-prone process. After extensive analysis we concluded that it is impossible to predict how the different concerns are specified. To make it possible to define architecture transformations, we require the state machine diagrams in a normalised form. The associated restricted source metamodel only defines a subset of the models associated with the original source metamodel. For this, UML allows specialisation of its metamodel by *stereotypes* with *tagged values* and well-formedness rules or *constraints* that can be defined using the Object Constraint Language (OCL). Together, these allow for the definition of a UML *profile*.

5. Target view: TRS-based SMC

The target model of the migration is based on task-resource systems (TRS). Here, *tasks* correspond to manufacturing activities and *resources* correspond to mechatronic subsystems. The alternatives, as discussed in Section 3.2, are specified in terms of tasks, resources, and *rules* that define (compound) *precedence* relations between tasks. Decisions with respect to those alternatives are typically taken run-time rather than design-time. In our case, the target view is based on a research prototype that ASML uses to study the applicability of new architectural paradigms [8] for its SMC systems.

In the TRS paradigm, a manufacturing request is translated into machine behaviour using two phases. First, during the planning phase, a scheduling problem is instantiated for the context of a manufacturing request. For this, the request is interpreted through rules that operate on resource types (*capabilities*) and task types (*behaviours*). The first phase results in a digraph that consists of *tasks* (i.e., behaviour with a request context) and their relations. Second, a scheduling phase constructively assigns tasks to specific *resources* over time [8, 22]. This results in a fully timed, coordinated TRS that can be dispatched for execution.

The prototype implements the generic part (planner, scheduler, and dispatcher) of a product-line architecture for building TRS-based SMC-systems. This product-line architecture offers a generic re-usable solution, that can be tailored via domain-specific modules called system definition and subsystem interface [23]. These modules define the specific system under control and are amenable for code generation. In this paper, the view on the domain specific system definition is the target of the migration. To define this view, we introduce the metamodel in Figure 6.

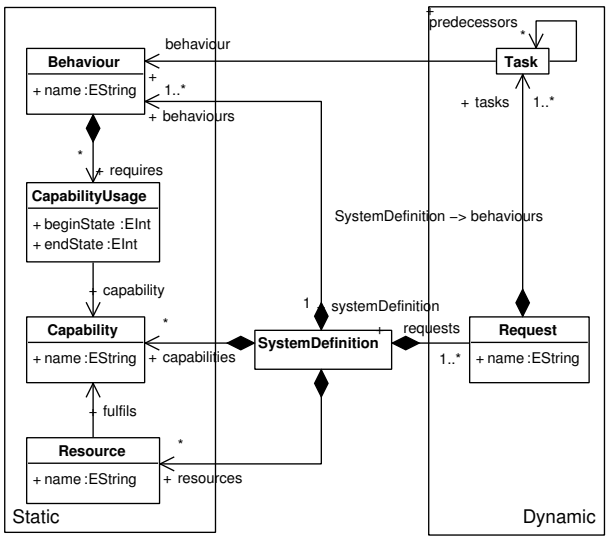
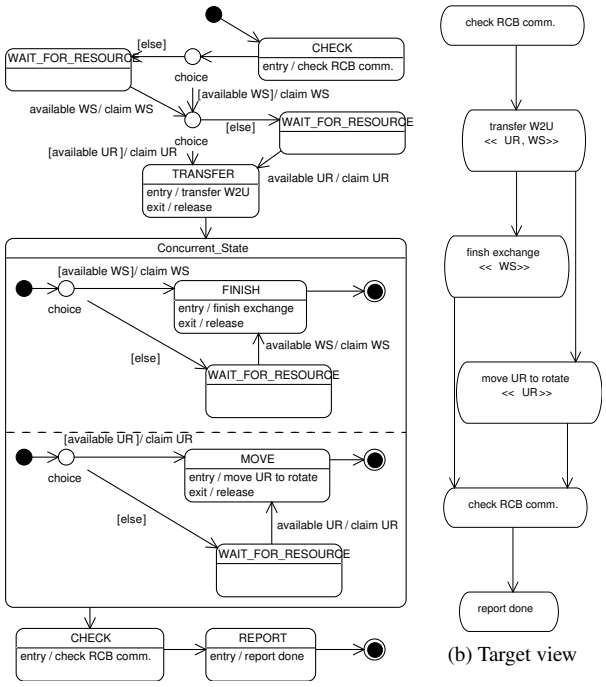


Figure 6: Target metamodel



(a) Normalised source view

(b) Target view

Figure 7: Unload wafer request transformation

The system definition contains a static and dynamic part and is represented in the TRS metamodel by the SystemDefinition, which serves as a root element. The static part defines the available Behaviours, Capabilities and Resources. In addition, it defines which Capabilities are used for which Behaviour and specifies the corresponding beginState and endState (CapabilityUsage). The dynamic part defines the rules for uniquely mapping a manufacturing Request to Tasks (that are of a specific Behaviour) and assigning Resources (that fulfil a required Capability).

As an example of how this metamodel can be used, consider the target view in Figure 7 of the previously introduced unload wafer request. Figure 7a) illustrates the normalised model of this request, which can be used as input for a transformation engine. Application of the transformation rules, as discussed in the sequel, results in the target model of Figure 7b). Note that we did not define a suitable graphical representation for TRS models yet. Hence, Figure 7b) mainly shows the dynamic part: a task digraph where nodes reference the corresponding behaviours. For convenience we added the required capabilities to the nodes between double brackets. Our transformation rules, however, also include the static part of the target model.

6. Transformation rules

In this section we discuss the transformation step of the FSM-TRS migration of SMC architectures. We first describe transformation rules for each of the elements of the target metamodel. Subsequently we describe rules for the concerns described earlier in the paper. These rules are generic with respect to the (FSM) source model.

All transformation rules in this section are implemented using the Atlas Transformation Language (ATL) [24]. An ATL transformation module consists of rules that contain a 'from' clause that matches elements in a source model and generates the corresponding elements in a target model as specified in its 'to' clause. The OCL-based syntax of ATL allowed us to express complicated patterns and mappings. In the illustrative ATL fragments, the metamodel elements used in the 'from' and 'to' clauses refer to the metamodels in Figure 4 and Figure 6, respectively.

The ATL transformation engine can be used in combination with XMI serialisations of models and metamodels defined using EMF or the MOF. Hence, we took the TRS metamodel from Figure 6 and defined it using EMF [25] and the MOF-UML metamodel available from the OMG [21]. To create the source model, we could simply use a UML modelling tool that supports XMI export. The engine then generates the target model in its serialised form. Using ATL we needed no more than 200 lines of code to implement all the necessary transformation rules.

6.1. Target-model instantiation

In general, the dynamic part of the target model can be derived directly and the static part is derived indirectly. The creation of the CapabilityUsage element is not discussed here as it is closely related to the resource usage concern.

SystemDefinition and Request The SystemDefinition root element in a target model, containing all required elements that define the domain specific part of an SMC controller, corresponds to a complete source model.

A Request invokes planning rules that determine how that particular manufacturing request, such as 'unload wafer' (Fig. 5), is planned. Planning rules involve a set of tasks and corresponding predecessor relations. In the source model, a state machine diagram is used to specify how a manufacturing request is to be executed. So, we create a Request element in the target model for every StateMachine in the source model.

Listing 1 shows the ATL implementation of this transformation rule. For now, we assume that a source model contains only a single StateMachine and thus the transformation rule can generate a `SystemDefinition` element for each `StateMachine`. It can be seen that `SystemDefinition` contains `behaviours`, `resources` and `capabilities`. However, these are created by other rules and ATL does not permit to navigate through the target model. Instead, we can navigate through the source model to the element that results in the creation of the desired target model element. For the case that a rule has multiple targets in its `to` clause, ATL offers the resolveTemp method. This method takes as input a source model element and a string that refers to a specific target of a rule. The string 't', for example, refers to the Task target of the Behaviour rule in Listing 2. Finally, a request is instantiated that contains a set of tasks which are also created by other rules.

```
rule SystemDefinition {
  from sm: UML!StateMachine
  to sd: TRS!SystemDefinition (
     behaviours <- thisModule.actionBehaviours()->
      collect(e|thisModule.resolveTemp(e,'b')),
     resources <- thisModule.actionResources()->collect(
      e|thisModule.resolveTemp(e,'r')),
     capabilities <- thisModule.actionResources()->
      collect(e|thisModule.resolveTemp(e,'c')),
     requests <- Set {rq}),
    rq: TRS!Request (
     name <- sm.toString(),
     tasks <- UML!Action.allInstances()->select(a|a.name
      ='Execute Behaviour')->collect(e|thisModule.
      resolveTemp(e,'t')))
}
```

Listing 1: Rule for SystemDefinition and Requests

Task and Behaviour In the original source view, a manufacturing activity is typically specified by an action and completion event. As an example, consider Figure 5, in which the action start transfer W2U and the event transfer W2U done result in a task that executes the behaviour transfer W2U (Fig. 7b). In the canonical source view, we assume run-to-completion semantics of actions and thus normalisation removes the completion events. Not all actions in the source view, however, correspond to manufacturing activities. Therefore, we only define a task in the target model for actions in the source model that correspond to a manufacturing activity, and a behaviour for every *type* of action.

As can be seen from Listing 2, we assume that `Tasks` and `Behaviours` in the target view correspond to `Actions` named 'Execute Behaviour' in the source view. The executed behaviour is specified in the `script` attribute. This also means that actions with identical `script` attributes effectively define an action type and should be mapped to the same behaviour. To implement this, we defined the `actionBehaviours()` helper function that first selects the set of actions named 'Execute Behaviour' and subsequently the set of actions with unique behaviours. For all actions in this set, a behaviour and task are created by the behaviour rule. Additionally we implemented a task rule that creates a task for all other actions named 'Execute Behaviour'.

A behaviour's `requires` attribute is set to a CapabilityUsage element. This is discussed in the explanation of the resource usage concern below. To set a task's corresponding behaviour we again use the `resolveTemp` construct. Finally, for tasks, the `predecessors` attribute has to be set to the set of direct predecessor tasks. This is also discussed later when we explain how the synchronisation and concurrency concerns are handled.

```
rule Behaviours {
  from a: UML!Action (
     thisModule.actionBehaviours()->includes(a))
  to b: TRS!Behaviour (
     name <- a.script.body,
     requires <- UML!SimpleState->allInstances()->select
      (s|s.entry=a)->asSequence()->first().incoming->
      collect(i|i.source)
       ->iterate(s; ss : Set(UML!SimpleState) = Set {}|
        thisModule.union(ss,s.getResourceClaims()))),
    t: TRS!Task (
     id <- a.script.body,
     behaviour <- thisModule.resolveTemp(thisModule.
      actionBehaviours()
       ->select(e|e.script.body=a.script.body)->
        asSequence()->first(),'b'),
     predecessors <- a.getPredecessors()->collect(e|
      thisModule.resolveTemp(e,'t')))
}
```

Listing 2: Rule for tasks and behaviours

This transformation partly relies on naming conventions. Rather than relying on naming conventions, we could introduce an ExecuteBehaviourAction as a specialised Action in the restricted source metamodel instead.

Resource and Capability In the FSM-based approach mechatronic subsystems were not modelled. Hence, in the source model there are no elements that directly correspond to resources and capabilities. We can, however, use the fact that, in the FSM-based approach, subsystems need to be explicitly claimed. We create Resources in the target model

based on actions that claim a specific subsystem. The implementation is straightforward and therefore not shown. To take into account that capabilities can be claimed multiple times for a request, we defined a helper similar to the one for the actions corresponding to a behaviour.

This transformation rule assumes additional constraints on the source metamodel. First, we require that claim actions are explicitly specified, which was often not the case. Second, we again rely on naming conventions for the recognition of claim actions, which we assume are named 'Claim Resource'. Similar to the situation with tasks we could again introduce a specialised Action instead.

6.2. Concern-based transformation rules

In this section we discuss how the transformation handles the various concerns that we have identified in Section 3.2.

Resource usage In the TRS based approach the CapabilityUsage element in the static part of the SystemDefinition specifies for every behaviour the capabilities it requires. Again, we cannot derive the CapabilityUsage elements in the target model directly.

In the original source models, various idioms were used to ensure that subsystems were used appropriately. Therefore, the normalisation step is particularly important when considering this concern. From Figure 5), and the corresponding normalised idiom from Figure 7a), it can be seen that additional actions are used for claiming the WS subsystem: if the WS subsystem is available it is claimed and the Transfer W2U action can be executed, otherwise a wait state is entered. This wait state is exited when an event is received that the WS subsystem has become available. Then, a claim action is executed in conjunction to ensure mutual exclusive use of subsystems. Finally, after the activity has been performed the claimed subsystems are released in a release action. This pattern can easily be generalised.

Thus, for each such pattern occurring before an action corresponding to a manufacturing activity and after the previous release action, we conclude that this activity requires the involved subsystems. In the target model CapabilityUsage elements are then defined connecting the involved corresponding behaviour and capabilities. In our example, the CapabilityUsage element relates the Transfer W2U behaviour to the WS capability.

In our target view, this results in the definition of a CapabilityUsage element relating the Transfer W2U behaviour to the WS capability. Again, for the normalised source view we assume a strict naming convention. This can be seen from the `from` clause of the rule in Listing 3, it matches all states named `'WAIT_FOR_RESOURCE'`. The `to` clause of this rule creates a CapabilityUsage element in the target view. The `resolveTemp` construct is used to set the `capability` attribute to the target of the rule that matches the 'Claim Resource' Action involved in the resource usage pattern.

Next, a behaviour is linked to CapabilityUsage elements by its requires attribute (Listing 2). This is done by first selecting all states directly preceding the state in which an action is executed that corresponds to the behaviour. On each of these predecessor states, we call the `getResourceClaims()` helper that recursively finds all states named `'WAIT_FOR_RESOURCE'` by reversing through the state machine until a release action is encountered. Here, we assume that a release action releases all claimed subsystems. The states in the returned set match the ResourceUsage rule and the Behaviour is linked by its requires attribute to the CapabilityUsage elements.

```
rule ResourceUsage {
  from
    s: UML!SimpleState (
     s.name='WAIT_FOR_RESOURCE')
  to cu: TRS!CapabilityUsage (
    capability <- thisModule.resolveTemp(thisModule.
     actionResources()->select(a|a.script.body=s.
     outgoing->select(t|t.effect.name='Claim Resource')
     ->asSequence()->first().effect.script.body)->
     asSequence()->first(),'c'))
}
```

Listing 3: Rule for resource usage pattern

Resource setups In the source view, resource state consistency is ensured by specifying setup transitions for every possible resource state at design-time. In practice, this is not done exhaustively. Instead, domain-knowledge is used to limit the number of setup related alternative transitions.

In the target view, setups are automatically inserted by the generic (solving) part of the product-line architecture. This is done at run-time, based on mismatching beginState and endState attributes of the CapabilityUsage element. To some extent, these could be derived from the explicitly specified setups in the source view. In this paper, we do not define a corresponding transformation rule as it depends heavily on domain knowledge. When reconsidering Figure 7, the move to rotate behaviour is in fact a resource setup that has been modelled explicitly to mimic existing behaviour exactly.

Synchronous execution In the source view, synchronisation between sequential actions can be achieved by specifying them as entry or exit actions with run-to-completion semantics. For concurrent actions, all sequences are typically specified exhaustively. The synchronisation between actions that belong to different requests can be done through external events. Here, wait states are typically used: wait for a resource, a timer, some action or data.

The target view defines precedence relations between those tasks that require synchronisation (within the same request). In principle, these relations follow from the execution order of the manufacturing activities and the corresponding actions within a normalised state machine diagram.

In the target view, (virtual) resources are used for external synchronisation. The implementation of the FSM-TRS transformation currently does not handle synchronisation across different requests. However, if we would modify the restricted source metamodel to include a special type of Event to denote external events, implementation hereof would become a straightforward exercise.

A predecessor relation is created for every task by searching for its set of (direct) predecessor tasks. For this we have defined several getPredecessors helpers that each operate on a different source metamodel element using polymorphism. So, for each task the getPredecessors helper is invoked on its corresponding Action. Depending whether the involved Action was either a transition effect or a state entry action, the getPredecessors helper is invoked on the transition's source state or all of the state's incoming transitions (iteratively). In the case of a source state we again use polymorphism to distinguish between different types of states: `CompositeState`, `PseudoState`, `FinalState`, and `SimpleState`.

As an example, Listing 4 contains the getPredecessors helper for SimpleStates. The general pattern used for these helpers is that if the state or transition itself contains an (entry or effect) action corresponding to a task, it is returned. If not, the `getPredecessors` is called iteratively on the preceding state or all incoming transitions.

```
helper context UML!SimpleState def: getPredecessors() :
  Set(UML!Action) =
  if self.entry->oclIsUndefined() then
    self.incoming->asSet()
    ->iterate(t; p : Set(UML!Action) = Set{} |
      thisModule.union(p,t.getPredecessors()))
  else if self.entry.name='Execute Behaviour' then
    Set {self.entry}
  else
    self.incoming->asSet()
    ->iterate(t; p : Set(UML!Action) = Set{} |thisModule
     .union(p,t.getPredecessors()))
  endif endif
;
```

Listing 4: Collect predecessors on SimpleStates

Concurrent execution When reconsidering the source view for our example (Fig. 5), it can be seen that concurrency of the move to rotate and finish exchange is modelled by the exhaustive specification of the alternative sequences. Although this is certainly not the most elegant manner, tool and language limitations typically impose these complex constructs. This particular idiom is commonly observed for actions that are related to a single request. Intuitively, the canonical pattern for concurrency would be a composite state with orthogonal regions. In practice, multiple requests are to be handled simultaneously. In turn, this results in a number of patterns executing concurrently.

Basically, the predecessor relation is the mechanism used in the target view to allow or disallow concurrency: if two tasks are not related by the transitive closure of the predecessors relation, they can execute concurrently. Now, these (potentially concurrent) tasks are executed as soon as their predecessors are finished and the required resources are available. In turn, this also implies that a task can have multiple (concurrent) predecessors.

We used the previously discussed `getPredecessors` helper on CompositeState to handle this situation. The code is rather complicated due to the many possibilities for finding predecessors from CompositeStates and therefore not shown here. When a CompositeState is concurrent, we take the union of the `getPredecessors` of its composing CompositeStates (iteratively). If it is not concurrent, we take the `getPredecessors` of the composing FinalState. Note that we assume that a state machine in this CompositeState has exactly one FinalState.

7. Evaluation

Supervisory machine control migrations The transformation rules we have defined operate on normalisations of FSM-based specifications of SMC systems. As such, they can be applied to FSM-TRS migrations of SMC systems without loss of generality. The normalisation procedure, on the other hand, is context specific, as it depends on the legacy modelling conventions.

Application of the generic migration process as presented in this paper, requires that the involved metamodels and transformation rules are made explicit. This increases the understanding of the implications (and difficulties) of a migration. Furthermore, the need for experts on both the domain and the TRS paradigm, is confined to the definition of the transformation rules and in a lesser extent to the normalisation step.

Product lines The type of migration as described in this paper is particularly interesting as it involves the introduction of a product-line architecture. The derivation of product-line members from legacy products using model transformations seems to be an elegant approach of introducing a software product line to replace a set of individually developed products (or components). However, such an approach is only feasible if the allowed variability for product-line members can be defined by a metamodel, such as the metamodel for TRS-based SMC systems in Figure 6.

Normalisation In practice, models are typically made as complete and accurate as is demanded by their application. These demands become more stringent when the models are used as input for automated processing, such as model transformations and code generation. As a result, the (context specific) normalisation step is crucial to our migration approach as it facilitates application of our model transformations in industrial contexts where source models are typically used for communication and documentation purposes only.

The complexity of this normalisation step depends on the number of constraints that the restricted source metamodel adds to the legacy source metamodel (if present). Fewer constraints make the transformation, which is typically automated, more complex. However, the normalisation step, which is typically executed manually, requires less effort. This trade-off suggests an approach that starts with a simple transformation that is gradually evolved to reduce the number of constraints required on the restricted source metamodel. This results in a complex automated transformation and a manual normalisation that requires as less effort as possible. Intuitively, the extraction of information from the legacy source models is typically incomplete due to, e.g., undocumented features and, as a result, some (domain) expert knowledge is still required to complete the migration.

Model driven architecture Our work bears similarities with the MDA. In the MDA, model transformations are applied to platform-independent models to obtain platform-specific models. Our transformation rules do not change the behaviour as specified by the legacy state machine diagrams. Therefore, the legacy source model describes, in a platform-independent way, the same behaviour as the associated target model combined with the generic part of the product line. Hence, the generic part can be seen as a platform whereas the target model of our transformation constitutes a platform-specific model. The essential difference with the MDA approach, however, is the purpose of the involved models and transformations. We apply model transformations in a *migration* context whereas the MDA applies them in a *development* context. The question whether it makes sense to apply our model transformations in a development context lies beyond the scope of this paper.

Tools and languages The success of the MDA vision depends partially on the ability of modelling, transformation and code generation tools to cooperate. Here, an advantage of ATL is that it accepts metamodels to be defined in either the MOF or EMF. Furthermore it supports XMI, which means that (in theory) most UML modelling tools can be used to create source models.

As such, MOF and its implementations (e.g.,EMF and MDR), UML, and particularly XMI play an important role. However, in practice the availability of different versions of these three specifications made it difficult to setup an appropriate tool chain. For instance, we could not use the latest version of our UML modelling tool (Poseidon for UML) because the UML metamodel it uses, was incompatible with the ATL transformation engine. Although we took the liberty of selecting tools that were able to cooperate, this will not always be possible in practice.

Another advantage of ATL is its OCL-based syntax. This allows people that haven been working with the UML metamodel, to understand and create transformation rules quickly.

8. Conclusions and future work

In this paper we have formulated the migration of an FSM-based SMC architecture to a TRS-based SMC architecture, as a model transformation problem. Our experiments show that application of model transformations increases the understandability of such a migration and reduces the need for domain experts.

We have demonstrated that the MDA development framework can be applied in a migration context as well. However, a normalisation step is required to overcome semi-formal, incomplete and ambiguous specifications as well as tool and language limitations. We have identified a trade-off between the inherent complexity of automated transformations and the required manual effort during normalisation. The introduction of the normalisation step makes our approach suitable for industrial applications.

Based on SMC-specific concerns and a normalised view, we have defined and implemented a set of generic transformation rules that support a migration towards TRS-based product-line architectures. The applicability of these rules has been illustrated for a real-world industrial case.

As such, the main contributions of this paper are:

- The extension of the MDA framework with a normalisation step to allow for migrations in industry.
- A set of transformation rules that can be applied to FSM-TRS migrations of SMC systems without loss of generality.

We are in the process of extending our work along the following lines. First, we want to apply our migration approach to other SMC components within ASML and further evaluate and quantify our approach. Second, we are investigating the possibility of deriving a normalised source model directly from the source code, thus minimizing the need for human intervention during the normalisation step.

Acknowledgements

The work in this paper has been sponsored by Senter (Ideals project) and NWO Jacquard (Reconstructor project). We would like to thank the Ideals team and ASML, in particular Remco van Engelen, Ed de Gast, Koen van der Heijden, Barend van den Nieuwelaar, Joost Worms and Asia van de Mortel- Fronczak for their input and feedback.

References

[1] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[2] R.J. Bril, R.L. Krikhaar, and A. Postma. Architectural support in industry: a reflection using C-POSH. *Journal of software maintenance and evolution: research and practice*, 17: 3–25, 2005.

[3] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[4] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[5] P. Gohari and W.M. Wonham. Reduced supervisors for timed discrete-event systems. *IEEE Transactions on Automatic Control*, 48(7):1187–1198, 2003.

[6] I. Sabuncuoglu and M. Bayiz. Analysis of reactive scheduling problems in a job-shop environment. *European Journal of operational research*, 126:567–586, 2000.

[7] G.C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 2002.

[8] N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[9] Spyros A. Reveliotis. *Real-Time Management of Resource Allocation Systems. A Discrete Event Systems Approach*, volume 79 of *International Series in Operations Research & Management Science*. Springer-Verlag, 2005.

[10] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings IEEE/IFIP Working Conference on Software Architecture (WICSA'04)*. IEEE Computer Society, 2004.

[11] IEEE-1471. IEEE recommended practice for architectural description of software intensive systems. IEEE Std 1471–2000, 2000.

[12] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures:Views and Beyond*. Addison-Wesley, 2002.

[13] Bas Graaf, Marco Lormans, and Hans Toetenel. Embedded software engineering: state of the practice. *IEEE Software*, 20(6):61–69, November–December 2003. Special issue on the State of the Practice of Software Engineering.

[14] OMG. MDA. `http://www.omg.org/mda`, 2005.

[15] Hoda Fahmy and Richard C. Holt. Using graph rewriting to specify software architectural transformations. In *Proceedings of Automated Software Engineering*. IEEE Computer Society, 2000.

[16] Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. IEEE Computer Society, 2000.

[17] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[18] Jan Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.

[19] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*. Springer-Verlag, 2002.

[20] Edward D. Willink. UMLX: A graphical transformation language for MDA. In *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA2003)*, 2003.

[21] OMG. OMG unified modeling language specification, version 1.4. `http://www.omg.org/technology/documents/formal/uml.html`, September 2001.

[22] Gérard Xavier Viennot. Heaps of pieces, I: Basic defintions and combinatorial lemmas. In *Proceedings of the Colloque de combinatoire énumérative (UQAM 1985), Montreal, Canada*. Springer-Verlag, May 1986.

[23] Bas Graaf, Sven Weber, and Arie van Deursen. Migration of supervisory machine control architectures. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*. IEEE CS, November 2005.

[24] ATLAS Group. The ATL home page. `http://www.sciences.univ-nantes.fr/lina/atl`, 2005.

[25] Eclipse Foundation. Eclipse modeling framework (EMF). `http://www.eclipse.org/emf`, 2005.