Model-driven Migration of Supervisory Machine Control Architectures ⋆

Bas Graaf [a,*], Sven Weber [b,c] and Arie van Deursen [a,d]

[a] *Delft University of Technology, Delft, The Netherlands*
[b] *ASML, Veldhoven, The Netherlands*
[c] *Eindhoven University of Technology, Eindhoven, The Netherlands*
[d] *CWI, Amsterdam, The Netherlands*

Abstract

Supervisory machine control is the high-level control in advanced manufacturing machines that is responsible for the coordination of manufacturing activities. Traditionally, the design of such control systems is based on finite state machines. An alternative, more flexible approach is based on task-resource models. This paper describes an approach for the migration of supervisory machine control architectures towards this alternative approach. We propose a generic migration approach based on model transformations that includes normalisation of legacy architectures before their actual transformation. To this end, we identify a number of key concerns for supervisory machine control and a corresponding normalised design idiom. As such, our migration approach constitutes a series of model transformations, for which we define transformation rules. We illustrate the applicability of this model-driven approach by migrating (part of) the supervisory control architecture of an advanced manufacturing machine: a wafer scanner developed by ASML. This migration, towards a product-line architecture, includes a change in architectural paradigm from finite state machines to task-resource systems.

1. Introduction

As software systems evolve they tend to become increasingly complex [Lehman and Belady, 1985]. 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 [Bril et al., 2005].

In practice, increased complexity and loss of conformance between the architecture as intended and the architecture as implemented make a system more difficult to change [Perry and Wolf, 1992]. This results in an increase of both development and maintenance effort. The involved effort can, for instance, be reduced by the separation of concerns, the use of product-line architectures, model-driven development and automatic code generation.

In this paper we consider the migration of supervisory machine control (Smc) architectures towards a product-line approach that, amongst others, supports model-driven development and code generation. In practice, adopting such techniques requires architectural changes. When migrating towards a product line, such a migration needs to be applied repeatedly to migrate different product versions into product-line members. Therefore, ideally, one would like to make such a migration reproducible by automatically transforming one architecture into another. In this paper we investigate how this can be done using model transformations. Developing a model-driven migration approach is particularly beneficial in a setting where product migration is not a one-off exercise.

In an advanced manufacturing machine, supervisory control [Ramadge and Wonham, 1987; Gohari and Wonham, 2003] is responsible for the coordination of the (discrete) high-level machine behaviour. This requires, amongst others, interpretation of manufacturing requests, synchronisation, scheduling, conditional execution, and exploitation of concurrency with respect to the resulting manufacturing activities [Sabuncuoglu and Bayiz, 2000; Buttazzo, 2002; Reveliotis, 2005]. For advanced manufacturing machines, the control systems have an indicative order of magnitude of 10 Smc components, each encompassing $10^4 - 10^5$ lines of code.

⋆ This is a substantially revised and expanded version of our paper: Migrating Supervisory Control Architectures Using Model Transformations. In *Proc. 10th European Conf. Software Maintenance and Reengineering (CSMR 2006)*, IEEE CS, 2006.

* Corresponding author
Email addresses: b.s.graaf@tudelft.nl (Bas Graaf), sven.weber@asml.com (Sven Weber), arie.van.deursen@cwi.nl (Arie van Deursen).

This paper was motivated by the prototype migration of the SMC architecture of a wafer scanner as developed by ASML, a manufacturer of equipment for the semiconductor industry. We use this wafer scanner 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, separates concerns, and supports run-time dependent decisions [Van den Nieuwelaar, 2004]. As a result, the maintainability and flexibility of the migrated software systems is improved.

We consider the start and end point of the migration as different architectural views [IEEE-1471, 2000]. We refer to these views as the source and target view respectively. An important element of an architectural view is its primary presentation [Clements et al., 2002], which typically contains one or more diagram(s). In this paper we focus on the models and their governing meta-models underlying those diagrams. In our migration approach we use these models to consolidate and reuse as much existing design knowledge as possible. As such, we consider migration to constitute a series of model transformations, which we implemented using the Model Driven Architecture (MDA [OMG, 2005a]). It should be noted that we only consider the actual migration approach; the paradigms for the migration start point and end point are prescribed by our industrial case.

In order to define a reproducible mapping and perform the migration, we define practical transformation rules in terms of patterns associated with the source and target meta-models. These transformation rules are practical in the sense that they are based on an actual migration as performed manually by an expert. Based on this migration, we have formulated generic, concern-based transformation rules. These rules are defined using a model transformation language making our approach automated. Due to practical reasons, which are mainly associated with the informal use of modelling languages in industry [Graaf et al., 2003; Lange et al., 2006], we first normalise the legacy models before applying our model transformations.

Although we focus on the migration of the SMC architecture of a particular manufacturing system, the ASML wafer scanner, the contributions of this paper are applicable to similar (paradigm) migrations of supervisory control components in general. The presented industrial results serve as a proof a concept, additional migrations have to be performed before the results can be properly quantified. The experiences as outlined in this paper are, to a lesser extent, relevant for all software architecture migrations that can be seen as model transformation problems.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3 we introduce SMC, concerns specific to SMC systems, and our running example. A generic migration approach, which we use for the migration between the introduced architectural paradigms, is presented in Section 4. The source paradigm of the migration and the normalisation of its associated views are discussed in Sections 5 and 6. The target paradigm and our transformation rules are treated in Sections 7 and 8. We illustrate each step of the migration by means of a running example. Section 9 reflects on the migration results. Finally, we conclude in Section 10 with a summary of contributions and an overview of future work.

2. Related work

The process that we propose considers migration as a mapping from a source to a target view. This approach is inspired by the approach for architecture reconstruction as described by Van Deursen et al. [2004]. There, architecture reconstruction is considered to be a mapping from a source view that is extracted from code to an architectural target view.

Our process can also be seen as the application of the MDA to software *migration* rather than to software *development*. In the MDA, software development is conceived as a series of transformations from source models to target models. As such, in both processes, model transformations are applied but in our case an essential normalisation step is added to the original MDA framework.

Fahmy and Holt [2000a,b] discuss several types of generic architecture transformations that can be viewed as graph transformations. In this paper we consider domain-specific transformations on architectural models that are more complex than typed graphs; next to typed nodes, our models also include attributes on nodes and edges. Moreover, their transformations are intended for small, evolutionary changes to a software architecture, whereas the transformations as discussed in this paper are driven by the migration to a different architectural paradigm.

Bosch and Molin [1999] use architecture transformations during architecture design to realise the non-functional quality requirements of a system. Of the transformation types they identify, the application of an architectural style is closest to our work. To some extent, changing the architectural paradigm from FSM's to TRS's, as considered in this paper, could be understood as such a transformation. In our case, however, this transformation also results in a product-line architecture.

In other work, transformations are applied to the migration of software at the level of source code. Baxter et al. [2004] present a toolkit that uses generalised compiler technology for this purpose. Gray et al. [2004] use this toolkit for model-driven program transformations where vertical and horizontal transformations are identified. Here, vertical transformations concern the creation of software artifacts from artifacts at different abstraction levels (translation). Application of the MDA typically involves vertical transformations, whereas they investigate its applicability to horizontal transformations. The architecture migration we discuss can also be considered a horizontal transformation. However, where they focus on the source code, we consider migration at the design level.

3. Migration context

In this section we first define the SMC context. Next, we introduce the motivating case and running example for this paper: a typical wafer scanner as produced, for instance, by ASML. In this setting we briefly discuss the key concerns for SMC systems in general. These concerns need to be addressed during architecture migration. As such, they form the basis for the design of our normalisation and transformation rules.

3.1. *Supervisory machine control*

The machine control context is clarified in Figure 1. From a supervisory perspective, (sub)frames, transducers and associated regulative controllers form mechatronic subsystems that execute manufacturing activities to 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, to interpret manufacturing requests and to ensure feasible machine behaviour, a supervisory machine control component is required to coordinate the execution of manufacturing activities [Ramadge and Wonham, 1987; Sabuncuoglu and Bayiz, 2000; Van den Nieuwelaar, 2004].

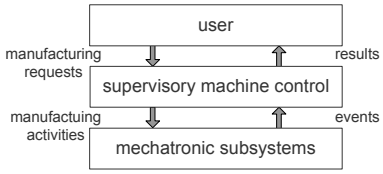


Figure 1. Machine control context

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 1: manufacturing activities of one level become manufacturing requests for the next level until the level of the mechatronic subsystems.

3.2. *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 2 illustrates a scanner and its subsystems.

A neighbouring machine, the track (TR), performs preprocessing steps and delivers silicon wafers to the prealignment system (PA), where the wafer orientation and alignment are determined and adjusted. Next, the load

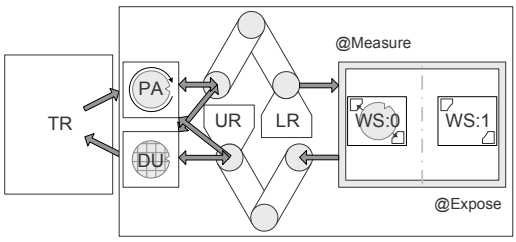


Figure 2. Simplified layout of a wafer scanner

robot (LR) transports the wafer to one of the two wafer stages (WS:0 or WS:1). Here, the wafer characteristics are measured. 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 the process used in 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.

For the SMC component of the wafer scanner depicted in Figure 2, we can identify the 'process wafer w' manufacturing request, which supports concurrent measuring and exposing of two wafers. To perform this request manufacturing activities such as 'load wafer w onto wafer stage WS:0' and 'unload wafer w from wafer stage WS:1' are executed. For instance, after a wafer has been exposed, and the stages have swapped, the wafer must be unloaded from its stage. In turn, these activities are requests for a lower-level SMC component. In this paper we will use the 'process wafer' and 'unload wafer' requests as illustrative examples.

3.3. *Concerns for supervisory machine control systems*

In advanced manufacturing machines, multiple manufacturing activities - and sequences hereof - may fulfil a particular 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 manufacturing activities and mechatronic subsystems to fulfil a given manufacturing request. For instance, when considering Figure 2, removing a wafer from DU can be done using either UR or LR. For supervisory control of advanced manufacturing machines in general, the following key concerns are identified.

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 of consecutive state

transitions. When these states do not match, an additional transition, a setup, has to be executed between consecutive activities. For instance, when UR is idle at PA, a rotation has to be performed before a wafer can be unloaded. In SMC, these *sequence-dependent setups* are common.

Intuitively, controlled *usage* of mechatronic subsystems is another important concern. 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 our wafer scanner example, the unloading of a wafer requires both UR and, for instance, WS:0.

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 between manufacturing activities or usage of the required mechatronic subsystems. In our wafer scanner example, one wafer can be measured and prepared for exposure while another wafer is being exposed.

Synchronous execution is another common concern. This not only refers to synchronisation of activities such that they are executed one after the other (e.g. load a wafer before processing it). It also applies to synchronisation of specific subsystem state transitions related to two activities. For instance, 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 (e.g. swapping WS:0 and WS:1).

Finally, *conditional execution* of manufacturing activities needs to be supported. That is, depending on certain conditions in a machine, different execution paths for a manufacturing request might be activated, each consisting of consecutive manufacturing activities. An example of such a condition in our wafer scanner example is the presence of another wafer on the wafer stage at the measure-side.

During migration, sequence-dependent setups, subsystem usage, concurrent execution, synchronous execution and conditional execution are concerns that need to be addressed. To this end, we defined concern-based transformation rules that map these concerns from the legacy to the new architecture.

4. Model-driven migration

Ideally, the migration of software architectures is complete, reproducible, reliable and automated. We consider the start and end point of the migration as different architectural views, referred to as the source and target view respectively. This is similar to the approach for architecture reconstruction as described by Van Deursen et al. [2004]. An architecture view is associated with a viewpoint [IEEE-1471, 2000], that, amongst others, specifies a meta-model for models underlying the primary presentation [Clements et al., 2002] of that view. In this paper we focus on those models.

For the migration of source models into target models we propose the migration approach as shown in Figure 3. It uses a two-step process that includes a normalisation and transformation step.

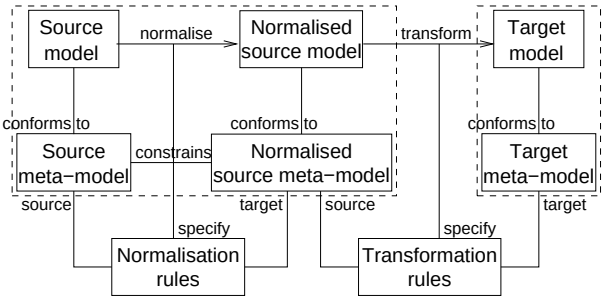


Figure 3. Generic two-phased migration approach

Models and their specifications are often incomplete and have a tendency to become inconsistent and ambiguous over time. This makes directly translating a source model into a target model inherently difficult. This is amplified further by tool limitations and the generally informal use of modelling paradigms and languages in industry [Graaf et al., 2003; Lange et al., 2006]. Combined with incomplete or generic meta-models (e.g. the UML meta-model), or no explicit meta-models at all, a multitude of models becomes conceivable that all have the same intended meaning.

In fact, an analysis of how SMC concerns are addressed in the source models for our migration, revealed a large variation in the used idiom. This makes it infeasible to specify generic corresponding transformation rules. As such, we introduce an intermediate normalisation step that uses a set of normalisation rules to obtain a normalised source model. The normalisation rules are defined as mappings from the source meta-model to the normalised source meta-model. This normalised meta-model describes a subset of the models described by the source meta-model. Next, a set of transformation rules can be applied to transform a normalised source model into the target model. These transformation rules are defined as mappings from the normalised source meta-model to the target meta-model.

In all, we see migration as a series of automated model transformations that are defined on meta-models to transform a source model into a target model using a distinct normalisation step. This approach is generic in the sense that it can be applied to any conforming source and target model without loss of generality. To actually implement this approach we require (normalised) source and target meta-models, normalisation rules, and transformation rules.

Although the approach is generic, our industrial case imposes some practical restrictions on the enabling technologies. Spurred by the fact that the existing architecture documentation contained source models (partly) in UML statecharts, we decided to implement the different steps of our

migration approach using MDA technologies. In the MDA vision, software development is considered to be a series of model transformations. Similarly, we consider software migration as a series of model transformations. Starting from UML, technologies compatible with MDA offer convenient and off-the-shelf means to define and manipulate models. Furthermore, the MetaObject Facility (MOF [OMG, 2005b]) can be used for the definition of meta-models. Finally, various model transformation languages are available to define transformations.

We defined all transformations in the Atlas Transformation Language (ATL), described by Jouault and Kurtev [2005]. An advantage of ATL is its syntax, which is similar to that of the Object Constraint Language (OCL). This allows people that have been working with the UML meta-model to understand and create transformation rules with relative ease. The actual ATL transformation engine relies on two implementations of MOF: the Eclipse Modeling Framework (EMF [Eclipse Foundation, 2005]) and the Metadata Repository (MDR [MDR, 2006]). The ATL transformation engine can be used in combination with XML Metadata Interchange serialisations (XMI) of models and meta-models that were defined using MOF. As our source meta-model we used the MOF-UML meta-model available from the OMG [OMG, 2001]. To create source models, we can simply use a UML modelling tool that supports XMI export. For the target meta-model we also used EMF as it allows for automatic generation of a primitive, tree-based editor for any arbitrary meta-model. This editor can then be used to inspect the results of our transformations.

5. Source meta-model

In this paper, we consider FSM's as the given starting point for the migration. The use of FSM's as a paradigm for supervisory control has been proposed by, for instance, Ramadge and Wonham [1987]. 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 for all requests to be specified explicitly using (un)conditional state transitions with associated triggers (events), and effects or state actions (manufacturing activities). When using this paradigm, concurrent execution is the result of independent parts of concurrently executing state machines that can optionally share events to synchronise. Consequently, multiple FSM's are used per controller (typically one for each type of request).

Our source models are specified using UML statechart diagrams. The relevant part of the meta-model is shown in Figure 4. Apart from this meta-model, the UML specification also provides a large number of well-formedness rules, specified in OCL, of which a few are mentioned below. Using this meta-model, UML state machines can be constructed that model behaviour as a traversal of a graph

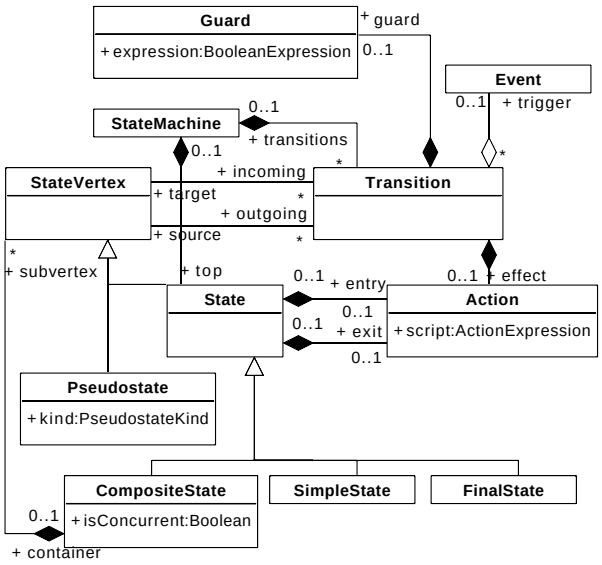


Figure 4. Source meta-model (excerpt from [OMG, 2001])

of state nodes interconnected by transition arcs.

In Figure 4 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 invariant over state variables holds. In addition, an optional entry or exit Action is executed when the state is entered or exited. The meta-model defines different types of States. A CompositeState contains (owns) a number of substates (subvertex). If a CompositeState is concurrent (isConcurrent) it contains at least two composite substates that execute in parallel. A SimpleState is a State without any substates. Execution of an enclosing CompositeState ends when a FinalState is entered.

Next to state nodes that describe a distinct situation, the meta-model also offers a type of StateVertex that models a transient node of a state graph: a Pseudostate. It allows modelling of more complex (conditional) transition paths. Three types of pseudo-states (PseudostateKind) are relevant for the state models in this paper: initial, choice, and junction. An initial Pseudostate is the default node of a CompositeState. A choice Pseudostate is used to create a dynamic conditional branch that depends on the action on its incoming transitions. Alternative paths may be joined using a junction Pseudostate.

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 an Event (trigger). A Transition without such an explicit trigger is fired by an implicit completion Event that is generated upon completion of all activities in the currently active State. A Guard is a boolean expression attached to a Transition that disables or enables its firing upon occurrence of its trigger (depending on whether it evaluates to true or to false). The effect of a Transition specifies an Action to be executed upon its firing. Finally, a StateMachine consists of a set of transitions and a top State that is a CompositeState.

As an example of how this meta-model is used in prac-

tice, consider the state machines in Figure 5, which correspond to the process wafer and unload wafer requests as introduced in Section 3.2. Such state machines are the source models for the migration. Note that our example requests were adopted from two distinct supervisory control components with an indicative order of magnitude of 10 requests, $10 - 10^2$ states, and $10^2 - 10^3$ transitions. Although we use actual manufacturing requests as running examples, we do not depict or discuss these requests in full detail for reasons of confidentiality.

From the number of choice pseudo-states and guarded transitions it becomes clear that conditional execution is the dominant concern in the process wafer request in Figure 5(a). In other words, the activated path is dependent on conditional synchronisation (e.g. wafer@measure) with other, concurrently executing requests.

Figure 5(b) illustrates that after the actual transfer (TRANSFER_FINISHED) the alternative completion sequences of subsequent activities, which are associated with the UR_moved and WS_moved events, are specified exhaustively. Furthermore, observe the use of two distinct resource usage patterns for WS and UR in our unload wafer request: for WS only an available Event (WS available) and release Action (release WS) are specified, for UR also a claim Action (claim UR) has been specified.

Note that, for reasons of simplicity, we choose not to include resource usage and setups in the specification of the process wafer request. Even from our example requests it becomes clear that, in practice, concerns are addressed using a multitude of idioms and constructs. This is the main reason for the introduction of our normalisation step.

6. Normalisation Rules

UML, as a generic modelling language, lacks constructs to support its application in the domain of SMC systems. This makes that, when using 'plain' UML, various design idioms are available for handling SMC concerns. For instance, guards (e.g. 'subsystem is available') were often modelled as events (e.g. 'subsystem becomes available') although these are fundamentally different. Similarly, manufacturing activities can be specified as actions on state transitions or as actions in separate states. This idiom diversity is fuelled further by tool limitations. For instance, tools that support a specific UML version, do not necessarily support all of its constructs.

In order to define architecture transformations, we need source models in a normalised form. These normalised models are associated with a meta-model that adds constraints to the legacy source meta-model and augments it with SMC-specific constructs. These constraints and additional model elements are used in well-formedness rules that prescribe how SMC-specific concerns are to be specified. For this, UML allows attaching constraints to model elements using OCL, and for the definition of additional model elements by stereotypes. Together, these enable the definition of a suitable UML-SMC profile for the normalised source meta-model. Example diagrams that conform to this profile are shown in Figure 6.

Table 1
SMC profile stereotypes

Stereotype	*baseClass*	*description*
≪wait≫	State	wait for resource state
≪claim≫	Action	claim resource action
≪release≫	Action	release resources action
≪available≫	Guard	resource available guard
≪available≫	Event	resource becomes available event

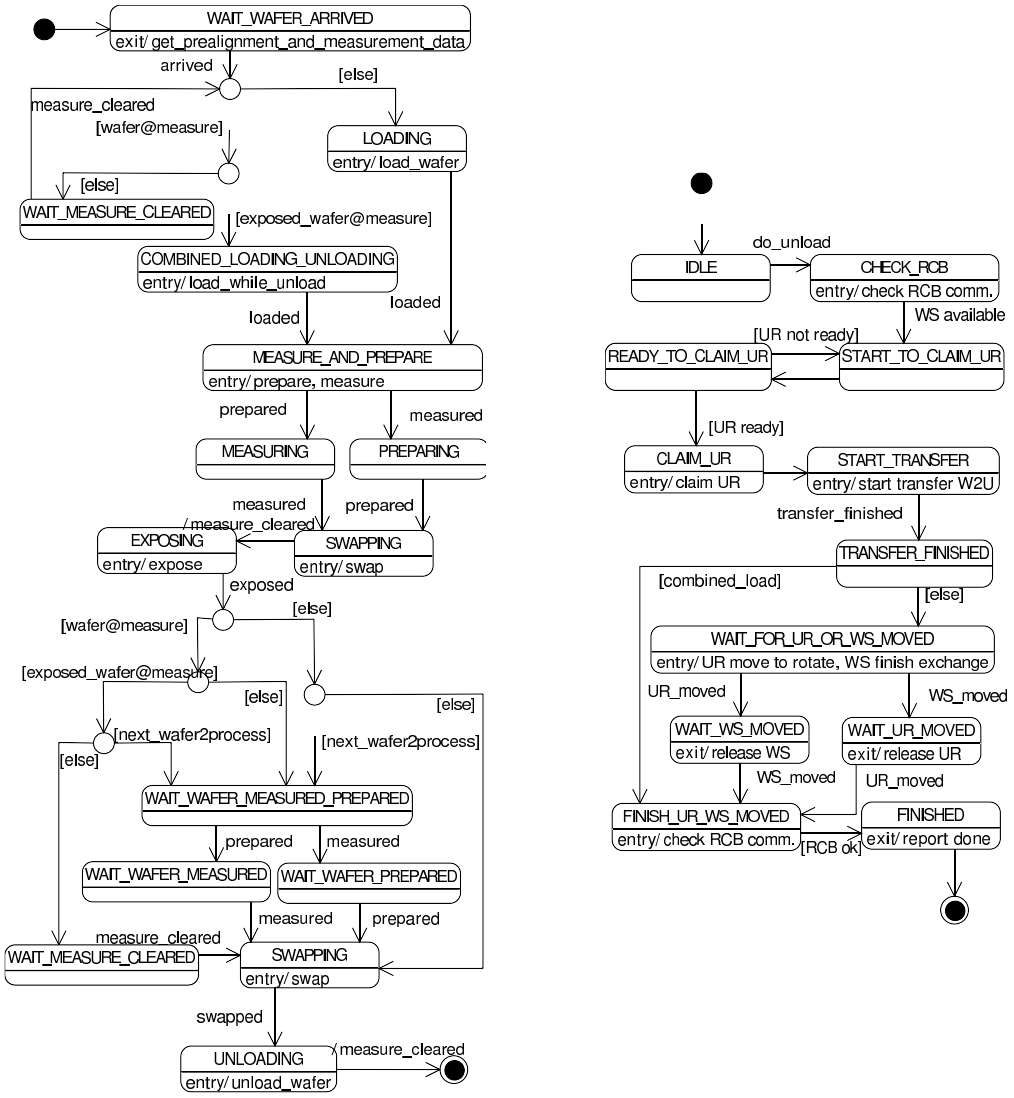
Normalised source models have to comply to a number of well-formedness rules. Most importantly, concerns have to be specified in a uniform way. We have defined a standardised idiom for the concerns as identified in Section 3.3. We introduce this idiom by example of Figure 6. Normalisation involves modifying source models to remove any violation of these well-formedness rules.

Table 1 lists the stereotypes that we define as part of the SMC profile. Next to stereotypes, the profile also defines a number of constraints. Listing 1 lists some of these constraints, specified in OCL as invariants over the UML metamodel (*C1-C4*). We merely use the constraints indicated by the **def** keyword to define extra properties on the elements mentioned in their **context**. This simplifies the specification of other constraints. Application of these stereotypes and constraints is discussed below.

Intuitively, the normalisation is context dependent and requires (some) domain knowledge. Moreover, the normalisation rules not only depend on the specific source paradigm but also on the modelling conventions as encountered in the specific (industrial) migration context. Therefore, we illustrate the normalisation step by defining the used context-specific normalisation rules for our case study.

Subsystem setups In the source model, subsystem state consistency is ensured by specifying setup transitions for every possible subsystem 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. Although subsystem setups can performed automatically using the TRS paradigm and, thus, do not need to be specified explicitly, we do preserve them during the normalisation step. This in fact ensures that the migrated control system mimics the behaviour of the legacy control system exactly. When reconsidering Figure 5(b) and 6(b), the move to rotate Action is in fact a resource setup.

For the normalised source model we do not use a specific idiom for setup activities; setups are modelled as any other manufacturing activity. If we would be less concerned with exact preservation of behaviour, setup activities could be simply removed during normalisation. In that case, domain knowledge is required to distinguish between setup activities and manufacturing activities.



(a) Process wafer

(b) Unload wafer

Figure 5. Example manufacturing requests

Subsystem usage The pattern to address the 'subsystem usage' concern is best understood from one of the orthogonal regions in the composite state in Figure 6(b). Before a manufacturing activity (e.g. finish exchange) that requires a certain subsystem (WS) is executed, a choice pseudostate is entered. Then, if the required resource is available ([WS available]), it is claimed (claim WS) by the transition towards the state in which the manufacturing activity is executed (FINISH). Otherwise, a state (WAIT_FOR_WS) is entered that is only left when an event occurs indicating the resource has become available (WS available). The resource is claimed (claim WS) on the transition triggered by that event. Once the manufacturing activity is performed, claimed resources are released again by a release action that is executed when exiting the state (release). This pattern can easily be generalised.

We use the stereotypes defined by the SMC profile (Table 1) to distinguish between Actions, Guards, Events, and States related to the use of subsystems and those related to the execution of manufacturing activities (to which no stereotypes are applied). Normalisation introduces stereotypes for specific model elements that are related to the subsystem usage concern. Furthermore, from Figure 5(b), and its normalised counterpart in Figure 6(b), it can be seen that additional model elements are introduced to complete the pattern described above. Note that in Figure 6 stereotypes are displayed only for states: this is a limitation of the UML tool we are using (i.e. 'Poseidon for UML').

Application of the stereotypes to source models requires domain knowledge to recognise the subsystem usage concern. This becomes apparent when reconsidering Figure 6(b). Here, WAIT_FOR_WS is a state in which the system waits for a subsystem to become available. This is intuitively different from the WAIT_WAFER_MEASURED states in Figure 6(a), where the intention is to specify that the system waits for a manufacturing activity to be completed. The ≪wait≫ stereotype is only applied to the former state.

For normalisation of source models we require that re-

```
context Action def:
    -- an action is a release action if a stereotype named 'release' is applied to it
    let isRelease : Boolean = self.stereotype->exists(s|s.name='release')

context State def:
    -- a state is a wait state if a stereotype named 'wait' is applied to it
    let isWait : Boolean = self.stereotype->exists(s|s.name='wait')

context Event def:
    -- an event is an available event if a stereotype named 'available' is applied to it
    let isRelease : Boolean = self.stereotype->exists(s|s.name='release')

-- C1: all release Actions are state exit actions
context Action inv:
    isRelease implies State.allInstances->exists(s|s.exit=self))

-- C2: a wait state has at least one outgoing transition triggered by an available event
context State inv:
    isWait implies outgoing->exists(t|t.trigger.isAvailable))

-- C3: state entry actions are actions that execute manufacturing activities (i.e. without stereotype)
context State inv:
    entry.stereotype->isEmpty

-- C4: all state nodes have no more than two incoming and outgoing transitions
context StateVertex inv:
    outgoing->size() <= 2 and incoming->size() <= 2
```

Listing 1. Some well-formedness rules of the SMC profile, in OCL

source usage patterns are made complete. In Figure 5(b), for instance, only a release action is specified for WS. In Listing 1 *C1*, *C2*, and *C3* are related to the subsystem usage pattern. *C1* specifies that a ≪release≫ Action only occurs as a state exit Action. *C2* states that at least one of the outgoing Transitions for a ≪wait≫ State is triggered by an ≪available≫ Event. Finally, to conform to constraint *C3*, all Actions related to manufacturing activities are moved to States as entry Actions. An example of this is the report done (entry) Action in Figure 5(b) that was normalised to an exit Action (Fig. 6(b)).

Synchronous execution Synchronisation between subsequent manufacturing activities in the source models is simply achieved by their order in the state machine. Furthermore, synchronisation between subsystem state transitions is not modelled at this level. As such, no specific idiom is used to specify this concern. In general, however, we have to take this concern into account while normalising the patterns associated with other concerns. While inserting and moving activities we have to make sure that we do not change their order in the normalised source model.

Concurrent execution In the original source models, concurrency was often modelled using States, including Actions that *start* two or more manufacturing activities and separate transition paths for all possible completion sequences, which are enabled by (external) completion Events. As an example, consider state MEASURE_AND_PREPARE and associated completions events prepared and measured in Figure 5(a). Because those events can only be associated with their corresponding manufacturing activities using naming conventions, such an approach complicates the determination of the scope of concurrent execution. Therefore, we require that concurrency is modelled using a concurrent CompositeState containing (orthogonal) regions. This implies that during normalisation, manufacturing activities are mapped to CompositeStates when they are started in a single State node and alternative completion sequences are specified exhaustively. Figure 6(b) contains an example of concurrent execution, where two resource usage patterns are executed in parallel.

Conditional execution The idiom for conditional execution is more complicated. First, we require it to be specified using a choice Pseudostate, having two outgoing Transitions. One specifies some condition as a Guard, while the other specifies [else] as a Guard. Furthermore, we require 'proper' nesting of conditional activation paths in a state machine. This means that we require pairs of corresponding, alternative paths through the state machine to be merged one at a time (using junction Pseudostates), and

in reverse order. Figure 6(a) contains several (nested) examples of this pattern.

Without this requirement for proper nesting, finding the set of States, and thus Actions, which are enabled when some Guard evaluates to true would become rather complicated. For the transformation of our source models to target models, finding this set of states is a necessary step. 'Non-proper' nesting occurs, for instance, in the bottom-half of the process wafer request in Figure 5(a). This results in replication of the activities performed on each path during normalisation. The three CompositeStates in the bottom-half of Figure 6(a) illustrate this replication. Part of this particular normalisation step is covered by constraint $C4$, which states that a path through a state machine can only split in two paths and that no more than two paths can be joined in a single state node. Because the OCL constraint to express proper nesting is rather lengthy, we did not include it here.

7. Target meta-model

We consider TRS as the given paradigm for the end-point of the migration. This end-point is based on a research prototype [Van den Nieuwelaar, 2004]. Using the TRS paradigm, a manufacturing request is translated into valid machine behaviour in two phases. First, upon arrival of a manufacturing request, a scheduling problem in the context of that request is instantiated during a planning phase. For this, the request is interpreted through rules that operate on capabilities (resource types) and behaviours (task types). Here, a manufacturing activity corresponds to a task and a mechatronic subsystem to a resource. The first phase results in a hierarchical digraph that consists of tasks and their (precedence) relations. Nodes in this graph can be composite to either denote a set of tasks that all need to be executed or to denote a set of tasks of which only one will be executed based on some condition. Second, a scheduling phase constructively assigns tasks in this digraph to specific resources over time [Viennot, 1986; Van den Nieuwelaar, 2004]. This results in a fully timed, coordinated TRS that can be dispatched for execution.

The end-point for our migration is a product-line architecture, of which Figure 7 displays the module view. In this architecture, the decisional responsibilities are assigned to three generic and reusable components: Planner, Scheduler, and Dispatcher. This product-line architecture offers variability with respect to tasks and resources and can be instantiated for a specific controller by implementing System definition and Subsystem interface modules. These modules define the specific system under control and implement the interfacing with lower-level components. The former is amenable for code generation, allowing for a reduction of software development time and effort.

In order to define our target models, we introduce a governing target meta-model as depicted in Figure 8. There, the system definition from Figure 7 is represented by the

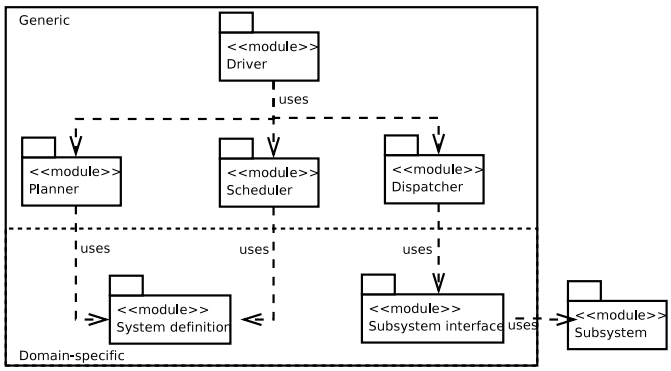


Figure 7. Module view for the product-line SMC architecture

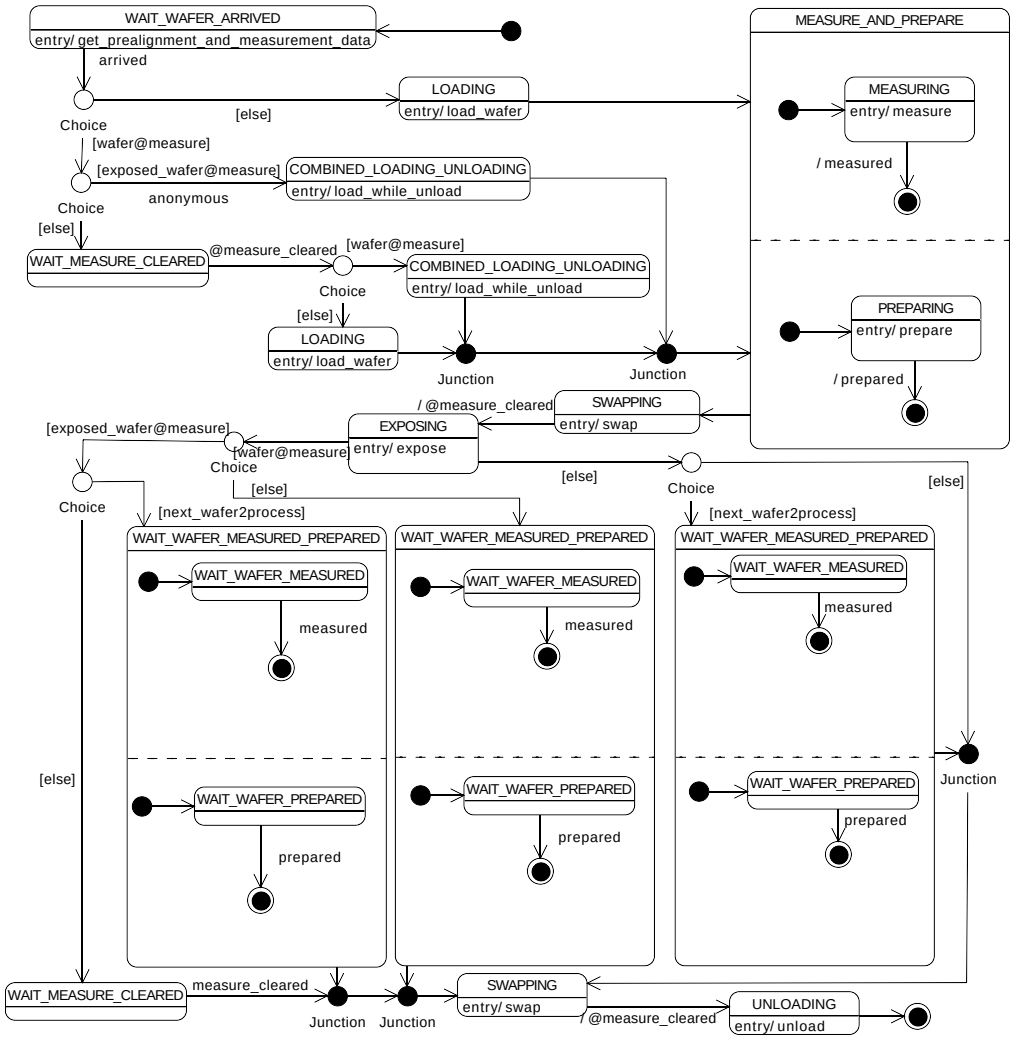
SystemDefinition, which serves as a root element. This system definition consists of a static and dynamic part. The static part defines the available Behaviours, Resources and Capabilities of the system under control. These are used to model types of manufacturing activities, subsystems, and types of subsystems. In addition, to address the subsystem usage concern, it defines which capabilities are required by which behaviour. Furthermore, the corresponding beginState and endState are specified in CapabilityUsage. These states are, for instance, used to determine sequence dependent setups.

The dynamic part of Figure 8 represents the rules for uniquely mapping a manufacturing Request to SimpleTasks, which are of a specific Behaviour, and assigning Resources that fulfil a required Capability. Conditional execution can be specified using OrTasks, that contains two Tasks (iftrue and iffalse) that may be composite. The evaluation of its condition determines which one will be dispatched. To cluster Tasks that *all* need to be performed, an AndTask can be used. Finally, every Task includes a set of predecessors, i.e. other Tasks that need to be executed before it can be dispatched. This relation is used to (dis)allow concurrency and imply synchronisation.

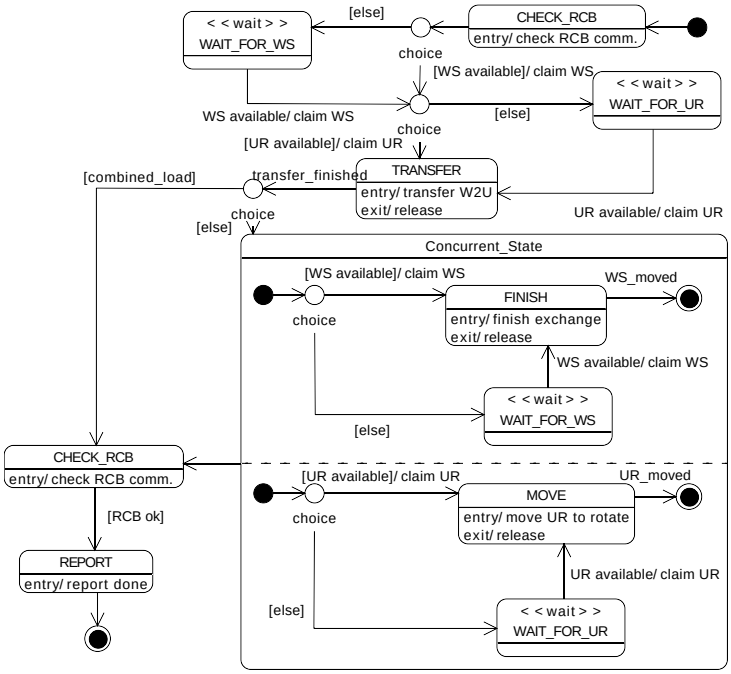
8. Transformation

Our transformation rules are defined as mappings from a normalised source meta-model (i.e. our UML profile) to a TRS meta-model. We used MOF to define the target meta-model rather than tailoring the UML using yet another profile. In this section we first introduce the transformation language that was used to define the transformation step of our migration approach.

For the definition of our transformations we used the following strategy. First, we indicate how elements in the normalised source meta-model are related to the primary elements of the target meta-model. Second, for each of the identified SMC concerns we define and tailor transformation rules to relate the corresponding patterns in the normalised source model and the target model. These rules are described reasoning backwards, meaning that for each of the elements of the target meta-model we explain for what source model patterns they will be created.



(a) Normalised process wafer request



(b) Normalised unload wafer request

Figure 6. Normalised source models

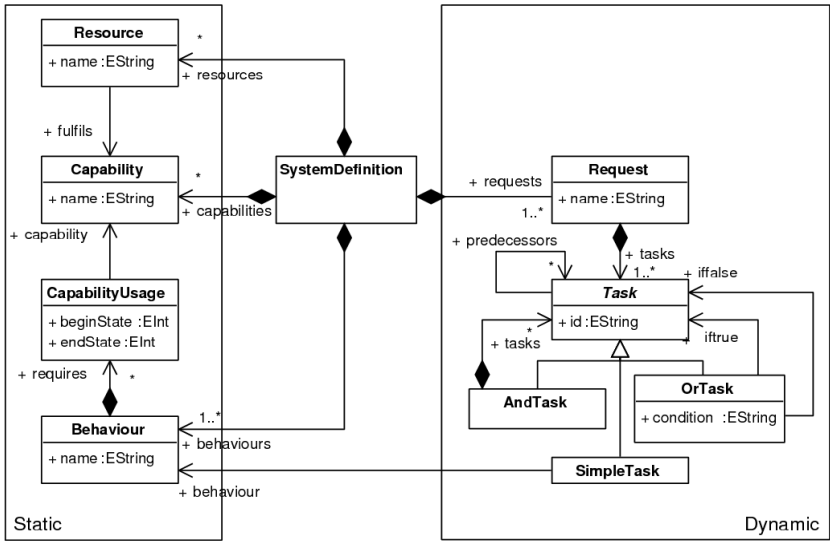


Figure 8. Target meta-model

In all, application of these transformation rules to a source model that conforms to our SMC profile, results in a target model that defines the System definition module for a particular SMC component (i.e. an instance of the architecture depicted in Fig. 7).

Next, we first introduce ATL. Then we discuss rules that generate the elements of the 'basic' types of the target meta-model in Figure 8: SystemDefinition, Behaviour, Capability, Resource, Request, and SimpleTask. Subsequently, we describe rules to create the elements and relations related to the concerns as previously described in Section 3.3. Finally, we discuss the results of the application of these rules to our example requests.

8.1. *The Atlas Transformation Language*

All transformation rules are implemented using ATL. As an example, consider the ATL fragment in Listing 2. An ATL transformation module consists of rules that contain a **from** clause, specifying a source pattern (s), and a **to** clause specifying a target pattern (t). The source pattern consists of a source type (UML!SimpleState) and an optional guard, which is a Boolean expression specified in OCL. The target pattern consists of a set of elements that each specify a target type (TRS!SimpleTask) and an associated set of bindings. A binding refers to a feature of the type (e.g. predecessors) and specifies an expression that is used to initialise the feature. The source and target types in the transformation rules in this paper refer to the source and target meta-models in Figures 4 and 8. As such, the rule in Listing 2 matches SimpleStates that conform to some constraints expressed by the guard. This rule generates a SimpleTask for which it specifies a set of bindings.

For every element in the source model that matches the source pattern of a rule, the elements specified by the target pattern are created in the target model. Note that in ATL, the source model is read-only and the target model is write-only. This can also be seen from Listing 2, where only the source model is navigated to initialise the features referred to in the bindings of the target pattern. Therefore, a specific value-resolution algorithm is used to initialise features: if the expression of a binding refers to another target element (created by the same rule) it is simply assigned, if it refers to a source element it is resolved by application of the rule that matches that source element and taking the default (first) target element.

```
rule Tasks {
    from s:UML!SimpleState (
        s.isTaskState and not thisModule.
          behaviourStates->includes(s))
    to t: TRS!SimpleTask (
        behaviour <- thisModule.resolveTemp(s.
          behaviourState,'b'),
        predecessors <- s.getPredecessors)
}
```

Listing 2. ATL example

For cases where the required target element is not the default element of another rule, ATL offers the 'resolveTemp' construct, a so-called helper operation. It takes a source model element and a reference to a specific target element of the matching rule as input parameters. In Listing 2, for example, this is done in the binding of the behaviour feature. In this case s.behaviourState evaluates to a SimpleState that is matched by another rule with multiple target elements, of which the 'b' target element is selected to bind to that feature.

Helpers are typically defined in the context of a meta-model element and effectively add a feature or operation to instances of that element (cf. the use of OCL definition constraints in List. 1). Alternatively, a helper can be defined without any context. Then, the default context of the complete transformation module, represented by the

thisModule element, applies. The resolveTemp helper is also defined in this default context.

8.2. *Basic target model elements*

SimpleTask and Behaviour SimpleTasks correspond to manufacturing activities, and Behaviours correspond to *types* of manufacturing activities in SMC systems. Therefore, to create SimpleTasks and Behaviours in the target model we need to identify Actions corresponding to manufacturing activities in the source model.

According to the UML SMC profile, an Action that corresponds to a manufacturing activity has no stereotype and is executed as a State entry Action (see `C3` in List. 1). For every such Action, a SimpleTask needs to be created in the target model. This is specified by the rule in Listing 3. It contains a guard that uses the behaviourStates helper to only match SimpleStates that map to a Behaviour. Note that in the specification, we do not map Actions to SimpleTasks, but instead we map the SimpleStates in which they are executed to SimpleTasks. This does not adversely affect our migration results since Actions corresponding to SimpleTasks are always State entry Actions (by constraint `C3` of the SMC profile).

In the source model, the executed behaviour is specified in the Action's script attribute. Therefore, Actions with identical script attributes effectively define an Action *type* and should be mapped to the same Behaviour. To implement this, the behaviourStates helper first selects the set of SimpleStates corresponding to a SimpleTask (i.e. all SimpleStates with entry Actions without stereotype) and subsequently determines the set of Actions with unique Behaviours. For all Actions in this set, a SimpleTask and a Behaviour are created by the behaviour rule. Additionally, we have implemented a rule that creates a SimpleTask for all other SimpleStates with such entry actions.

For (Simple)Tasks, the predecessors attribute has to be set to the set of direct predecessor tasks. Furthermore, a Behaviour's requires attribute is set to a CapabilityUsage element. This is discussed in Section 8.3 for the related synchronous execution and subsystem usage concerns.

Resource and Capability To create Resources and Capabilities we need to identify mechatronic subsystems in the source models. However, in the FSM paradigm, mechatronic subsystems are not modelled explicitly. Hence, the source model does not contain elements that directly correspond to Resources and Capabilities. We can, however, take advantage of the fact that in the FSM paradigm, subsystems are explicitly claimed. We create Resources in the target model based on Actions that claim a specific subsystem, i.e. Actions to which the ≪claim≫ stereotype has been applied. Furthermore, for every resource we simply create a separate Capability (Resource type).

In the specification of the involved transformation rules (not shown) we had to take into account that Capabilities can be claimed multiple times during a single request. This results in multiple Actions claiming the same Capability. Because we do not want to create a separate Capability for each of the Actions claiming the same capability, we defined a helper similar to the behaviourStates helper.

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

A Request encompasses rules that determine how that particular manufacturing request, such as our unload wafer from Figure 5(b), is planned. Planning rules involve a set of Tasks and corresponding predecessor relations. Additionally, a Task can be an AndTask or an OrTask. In the source model, a complete state machine 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 4 shows the ATL specification of this mapping. The Request rule generates a Request element for every StateMachine in the source model. This Request contains tasks which are created by other rules. As will be explained later, States or Guards in the source model may map to Tasks in the target model. Because the tasks in our target model may be composite in which case they *own* other tasks, we should take care not to select all model elements in the complete state machine that map to a Task. Instead, for a Request we discard all States or Guards inside a CompositeState other than the top, and on paths that are only conditionally enabled (i.e. by a transition's guard). To this end, we defined two additional generic helpers. First, rootOfSubTree takes a set of states as argument and recursively selects the 'first' state of that set (i.e. the one without incoming transitions from other states in the set). Second, getTaskModelElements is applied to that 'first' State to collect all model elements that map to a Task. In essence, this helper takes a set of states and traverses this set as a state 'tree' starting from the State (or Guard) it is applied to, and bypassing CompositeStates and conditional paths. During this traversal it collects all model elements it encounters that map to a Task (i.e. Guards or States).

The SystemDefinition rule generates a SystemDefinition element that corresponds to the complete source model. The behaviours, resources and capability features of the SystemDefinition element are bound to the result of other rules. In particular for behaviours and resources we had to use the resolveTemp helper as these are not created by the default target elements of the involved rules. In this case, the relevant source model elements are selected by two helpers that are defined in the context of the transformation module itself: behaviourStates gives all the source model elements (SimpleStates) that map to a Behaviour, and resourceActions gives all the source model elements (≪claim≫ Actions) that map to a Resource. The request

```
rule Behaviours {
    from s: UML!SimpleState (
        thisModule.behaviourStates->includes(s))
    to t: TRS!SimpleTask (
        behaviour <- b,
        predecessors <- s.getPredecessors),
    b: TRS!Behaviour (
        name <- s.entry.script.body,
        requires <- s.incoming->collect(i|i.source)->iterate(s; ss:Set(UML!SimpleState) = Set {}|ss->union(s.
          getResourceClaims)))
}
```

Listing 3. Rule for tasks and behaviours

```
rule Request {
    from sm: UML!StateMachine
    to rq: TRS!Request (
        tasks <- thisModule.rootOfSubTree(sm.top.subvertex,sm.top.subvertex->asSequence()->first()).
          getTaskModelElements(sm.top.subvertex))
}
rule SystemDefinition {
    from sm: UML!Model
    to sd: TRS!SystemDefinition (
        behaviours <- thisModule.behaviourStates->collect(e|thisModule.resolveTemp(e,'b')),
        resources <- thisModule.claimActions->collect(e|thisModule.resolveTemp(e,'r')),
        capabilities <- thisModule.claimActions,
        requests <- UML!StateMachine->allInstances())
}
```

Listing 4. Rule for SystemDefinition and Requests

feature is bound to the elements created by the Request rule for all StateMachines in the source model.

8.3. *Concern-based transformation rules*

Resource usage To address the resource usage concern we need to relate Behaviours to the Resources and Capabilities (resource types) they require. In the target meta-model, CapabilityUsage elements are used to this end. However, we cannot derive the CapabilityUsage elements in the target model directly, since our source models only contain dynamic information. Consequently, we will have to derive them indirectly instead.

For each subsystem usage pattern, as described in Section 6 we conclude that the subsystems claimed at that point are required for the corresponding manufacturing activity. These are all the subsystems that are claimed after the previous release action. In the target model, CapabilityUsage elements are then defined connecting the corresponding Behaviour and Capabilities. For our unload wafer request, for instance, this results in the definition of a CapabilityUsage element relating the transfer W2U behaviour to the WS capability.

The **from** clause of the rule in Listing 5 matches all ≪wait≫ States, using the isWait attribute helper. The **to** clause of this rule creates a CapabilityUsage element in the target model. The resolveTemp helper is used to set the capability attribute to the target of the rule that matches the ≪claim≫ Action involved in the resource usage pattern. Next, a Behaviour is linked to CapabilityUsage elements by its requires feature. Listing 3 shows that this is done by first selecting all States directly preceding the State in which an Action that corresponds to the Behaviour is executed. On each of these predecessor States, we iteratively call the getResourceClaims helper that recursively finds all ≪wait≫ States by backwards traversal of the state machine until a ≪release≫ Action is encountered. A ≪release≫ Action releases all claimed subsystems. The ≪wait≫ States in the returned set match the ResourceUsage rule and the Behaviour is linked by its requires attribute to the corresponding CapabilityUsage elements.

Resource setups In the target model, 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 model.

```
rule ResourceUsage {
    from s:UML!SimpleState (s.isWait)
    to cu: TRS!CapabilityUsage (
        capability <- thisModule.claimActions->select(a|a.script.body=s.outgoing->select(t|t.effect.isClaim).effect.
          script.body))
}
```

Listing 5. Rule for resource usage pattern

In this paper, however, we do not define a corresponding transformation rule as it depends heavily on domain knowledge. Using our transformations, setups will explicitly end up in the target model as just another task and behaviour. As said, this ensures that the migrated control system mimics the behaviour of the legacy control system exactly, thus resulting in a validated and acceptable baseline.

Synchronous execution The target model 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. In addition, (virtual) resources can be used for external synchronisation.

For synchronisation within a Request, predecessor relations are created for every task by searching for its set of (direct) predecessor tasks. For this we have defined two helpers that both operate on the elements that match rules that create Tasks. The first helper is depicted in Listing 6 and is defined on StateVertex whereas the second one is defined on Guard. For each Task, one of these getPredecessors helpers is invoked on its corresponding StateVertex or Guard. These helpers determine whether the current element (self) corresponds to a task. If so, this element is returned. Otherwise, the helper is recursively applied to the finite set of all direct preceding modelling elements that may map to a task.

Concurrent execution The normalised pattern for concurrency, as discussed in Section 6, is a CompositeState with orthogonal regions. To address the concurrent execution concern we need to identify instances of such patterns in the source model.

We defined a transformation rule that creates an AndTask for every concurrent CompositeState in the source model except for the top CompositeState of the StateMachine. Basically, the predecessors relation is the mechanism used in the target model to (dis)allow 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 execution of their predecessors has finished and the required resources are available. In turn, this also implies that a task can have multiple (concurrent) predecessors. Collecting predecessor tasks was already discussed in the previous paragraph.

Conditional Execution As discussed in Section 6, the normalised source model uses a state with two outgoing guarded transitions to specify conditional execution. Every two alternative conditional branches in a source model are mapped to an OrTask in the target model. This OrTask contains two subtasks (iftrue and iffalse), which may be composite and represent the two conditionally executed branches following a State with two outgoing guarded Transitions. Subsequently, for the creation of those subtasks, we need to find all model elements that map to a task in each of the branches.

The specification of this transformation rule is depicted in Listing 7. This rule matches one of the Guards (not the else) for every conditional execution pattern, determined by the isOrTaskGuard helper. It creates an AndTask for each of the two branches using the getTaskModelElements helper. The set of States that this helper uses to determine the scope in which it has to select all ModelElements that map to a Task is calculated by the guardedTaskStates helper. This helper selects all States 'guarded' by some guard. To this end, it calculates the difference between the path through the state machine that starts from the target of the conditional transition and the corresponding alternative transition path.

8.4. *Transformation results*

In total, we needed approximately 300 lines of ATL code to implement all the necessary transformation rules and helpers for the transformation step of our migration approach. Once the source model, source meta-model, target meta-model, and transformation module are defined and located, the ATL transformation engine generates the target model (e.g. a system definition) in its serialised form. The results as obtained for the normalised unload wafer request are depicted in Figure 9.

Figure 9(a) shows a screen capture of the created TRS target model, inspected using the tree-based editor that was generated for our TRS meta-model by the EMF plugin. There, TRS model elements are shown in a tree structure to indicate containment. Furthermore, it can be seen that we are dealing with a SMC component that accepts a Request unload_wafer. The selected element under the Properties tab in the bottom part reveals that "SimpleTask check RCB comm." can only be dispatched after its predecessor "OrTask combined_load" has been executed.

```
helper context UML!StateVertex def: getPredecessors:Set(UML!ModelElement) =
    if self.incoming->isEmpty() then
        Set{}
    else if self.isOrTaskStateJoin then
        self.getFork.outgoing->collect(e|e.guard)->select(e|e.isOrTaskGuard)
    else if self.incoming->collect(e|e.guard)->select(e|not e.oclIsUndefined())->exists(e|e.isOrTaskGuard or if e.
      oppositeGuard.oclIsUndefined() then false else e.oppositeGuard.isOrTaskGuard endif) then
        Set{}
    else if self.incoming->collect(e|e.source)->select(e|e.isTaskState)->isEmpty() then
        self.incoming->collect(e|e.source.getPredecessors)->flatten()
    else
        self.incoming->collect(e|e.source)->select(e|e.isTaskState)
    endif endif endif endif;
```

Listing 6. Collect predecessors on StateVertex

```
rule ConditionalExecution {
    from g:UML!Guard (g.isOrTaskGuard)
    to t: TRS!OrTask(
        condition <- g.expression.body,
        iftrue <- at_true,
        iffalse <- at_false,
        predecessors <- g.getPredecessors),
    at_true: TRS!AndTask(
        tasks <- g.transition.target.getTaskModelElements(g.guardedTaskStates))
    at_false: TRS!AndTask(
        tasks <- g.oppositeGuard.transition.target.getTaskModelElements(g.oppositeGuard.guardedTaskStates)
}
```

Listing 7. Rule for conditional execution patterns

The consequence of using a custom meta-model is that we only have the basic generated editor to visualise and document our transformation results. Again, we turned to model transformations to solve this problem. As there is no suitable graphical representation for complete TRS models yet, we defined a transformation that maps a TRS model to UML Activity Graphs for the dynamic part (one for each request) and a UML Class model for the static part. The result of this transformation can easily be displayed using UML tools. Figure 9(b), for instance, shows the dynamic part of our unload wafer request displayed as an UML Activity Graph.

9. Evaluation

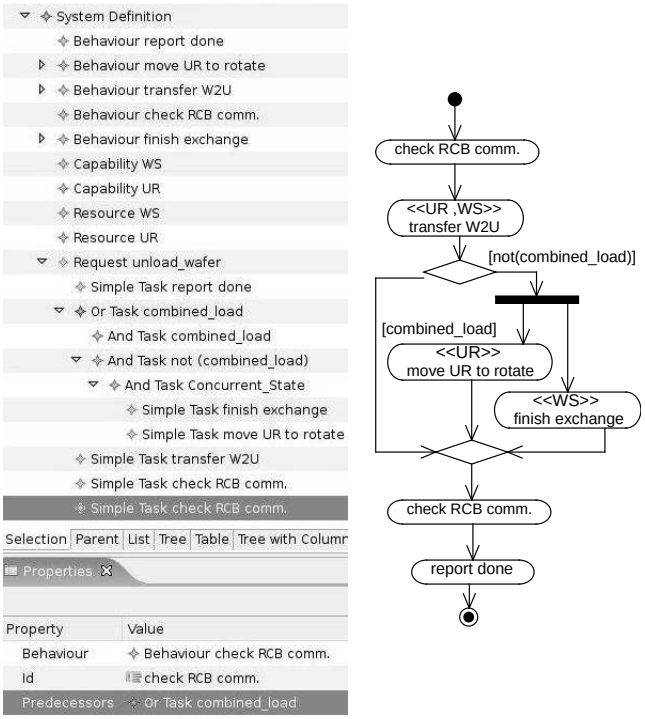
Applicability Application of our generic, model-driven migration approach requires that the source view and target view can be defined using a meta-model. When this is possible, the actual migration from source to target constitutes a series of model transformations.

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

MOF based meta-models only provide the abstract syntax for conforming models and do not define how to visualise them (concrete syntax). In fact this is a drawback of using a custom meta-model: no model editors and viewers are available, apart from the basic editor as generated by the EMF plugin. In this paper we again turned to model transformations to document and visualise our results. The use of model transformations provides an elegant and flexible way of generating architecture documentation that can easily be tailored to meet specific documentation requirements of a migration context.

It turned out that a model-driven migration approach based on MDA is useful for rapid (incremental) development of normalisation rules and transformation rules. That is, results can easily be visualised and documented given the wide variety of available tools.

Scalability With respect to the scalability of our approach we can safely state that our experiments are of the same order of magnitude as full-fledged component migrations for real-world wafer scanner applications. More concretely, the two requests that were migrated as a proof of concept



(a) TRS target model (b) Activity Diagram

Figure 9. Results for unload wafer request

account for approximately 10-20% of the source code for our SMC components. The application of our transformation rules to the two representative examples presented in this paper requires less than 10 seconds to complete on a modern notebook. Furthermore, we expect the execution time to be linear with respect to the number of requests. More important for the execution time is the nesting depth of conditional paths. For our industrial case we have not encountered requests with deeper nesting than our example requests.

Effectiveness Our model-driven approach requires that implicit design decisions and design knowledge is consolidated and made explicit for the definition of meta-models and transformation rules. This increases the general understanding of concerns and the associated implications (and difficulties) surrounding the architecture migration. Moreover, the need for experts on both the domain and the target paradigm is confined to the definition of the normalisation and transformation rules.

The effectiveness of both the MDA approach and our model-driven migration approach depends partially on the ability of modelling, transformation and code generation tools to cooperate. As such, standards involved with the MDA, such as MOF, UML, and particularly XMI, play an important role. In practice, the availability of different versions of these specifications made it difficult to setup an appropriate tool chain. For instance, we could not use the latest version of our UML modelling tool (i.e 'Poseidon for UML') because the UML meta-model it uses, was incompatible with the ATL transformation engine. Although we took the liberty of selecting tools that were able to cooperate, we still needed to implement some additional transformations using eXtensible Stylesheet Language Transformations (XSLT) to overcome some incompatibilities between the various tools. In industry it will not always be possible to select a specific set of tools for the migration given practical considerations such as licensing, support, and training costs.

Apart from tool support, the required human intervention during the normalisation step also determines the effectiveness of our migration approach. The complexity of the normalisation step depends on the number of constraints that the restricted source meta-model adds to the legacy source meta-model (if present). Here, a trade-off applies: fewer constraints make the transformation, which is typically automated, more complex because more specification alternatives have to be covered. For instance, if we would allow Actions corresponding to manufacturing activities to occur as Actions on Transitions, searching for predecessors would become much more complicated. On the other hand, the normalisation step requires less effort in that case.

In our case, the target meta-model specifies the domain-specific part of a product-line. We believe that model transformations are particularly applicable as a migration approach for the recurring migration of individual product-line members. In general, a model-based migration approach is beneficial in situations where a number of similar artifacts need to be migrated. Such a setting provides sufficient return on investment for the definition of meta-models, normalisation rules, and transformation rules.

More specifically, when considering the previously mentioned trade-off, a larger number of artifacts that need to be migrated justifies a higher investment in the definition of transformation rules, allowing for a less involved normalisation step. As another example of this trade-off, consider our assumption of proper nesting. It implies that alternative branches in a state machine are joined two at a time and in reverse order. One could relax this assumption (constraint) and implement a more intricate transformation rule to handle this relaxation.

Extensibility Currently, our transformation rules do not handle synchronisation across different requests. This could prove to be a limitation for the large scale application of our transformation rules. Hereto, we would have to (at least) extend our profile to include a special type of Event to denote external events for such inter-request dependencies.

The overall extensibility of our migration approach is demonstrated by using source models with two distinct origins for our experiments. In the case of the unload wafer request we used the available architecture documentation of the involved SMC component. This documentation contained UML statechart diagrams for the component's requests, including our example request.

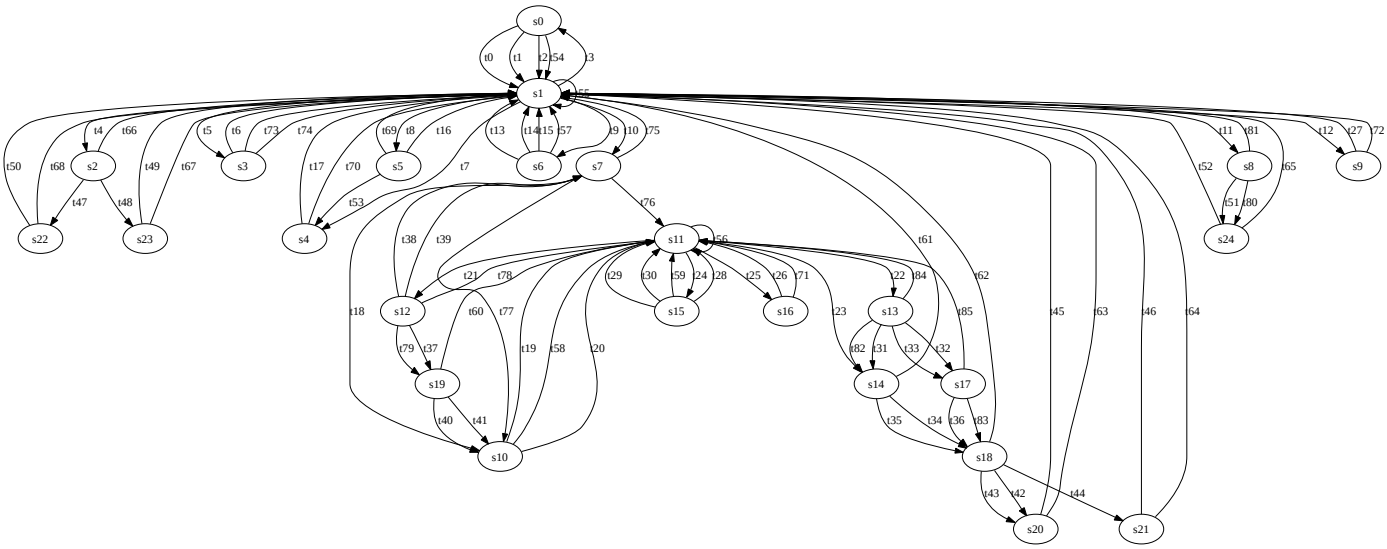


Figure 10. One of three concurrent state models (made anonymous)

However, for the SMC component that performs the process wafer request documentation was not available. Instead we had to reconstruct the source model from the source code. For this we took advantage of the fact that this component was based on a proprietary library for FSM's. Using this library, the component implemented three concurrent state machines that covered the combined behaviour of all requests and combinations hereof. Figure 10 depicts one of the component's three state machines.

This particular state machine illustrates the typical result of an evolving software architecture: two legacy state-based components were augmented with a new supervisor. This supervisor was obtained by taking the product of the two legacy state-machines and adding two choice pseudo-states (i.e. `s1` and `s11`) to allow for different activation paths, based on legacy request combinations.

We extracted the process wafer request state machine from the three implemented concurrent state machines and the corresponding source code by isolating state transition paths and combining them into a request state model. The resulting extracted source models were used as the input for our normalisation step. In fact, such an extraction step in which we isolate request state machines (i.e. to obtain models to be normalised) can be seen as an extension of the 'front-end' of our approach.

The 'back-end' of our approach can be extended as well by steps that further process the result of our model transformations. We already mentioned the generation of documentation. Another possible extension is the generation of source code to actually generate the System Definition module of the product-line architecture (Fig. 7). Both can be specified using model transformations.

Note that we did not yet consider the domain specific interface modules of the product-line architecture. However, this only constitutes a minor hurdle since we can simply encapsulate the existing source code bodies for each behavior (preserving interface functionality and behavior).

10. Conclusions and future work

In this paper we formulated the migration of SMC systems as a model transformation problem. The starting point is an SMC architecture based on FSM's; the end point is a product-line SMC architecture based on TRS's. Our approach supports the generic migration of the product-line members.

We demonstrated that the development framework for the MDA can be successfully applied in a migration context as well: migration can be seen as a series of model transformations. We proposed a generic two-phase model-driven migration approach that uses a distinct normalisation and transformation steps to derive the modules required to instantiate the TRS product-line architecture for a particular (sub)system. The normalisation step is crucial in overcoming semi-formal, incomplete and ambiguous specifications as well as tool and language limitations. This normalisation step requires domain knowledge and manual effort, but makes our approach suited for industrial application.

A trade-off has been identified between the inherent complexity of automated transformations and the required manual effort during normalisation. Based on SMC-specific concerns and a normalised source meta-model, 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. Since our transformation rules operate on normalisations, they can be applied to FSM-TRS migrations of SMC systems without loss of generality.

The industrial case that motivated this paper imposes not only the source and target paradigms but places practical constraints on the enabling technologies as well. Starting from UML, we selected technologies compatible with the MDA to setup a convenient tool-chain that supports the definition and manipulation of models. Using this tool

chain, several requests from different SMC components have been migrated as a proof of concept. Our experiments show that the application of model transformations not only increases the understandability of such a migration, but also reduces the need for domain experts.

As such, the main contributions of this paper are:

- The illustrated applicability of the MDA approach to architecture migrations. To this end, we introduced a vital normalisation step that enables migrations in an industrial setting.
- A practical view on the use of meta-models and profiles for migrations in general and, more specifically, on the normalisation, and transformation of SMC source models.
- The specification of a set of model transformation rules, an SMC UML profile, and a TRS meta-model that can be applied to FSM-TRS migrations of SMC architectures.

We are in the process of extending our work along the following lines. First, we want to further investigate the extraction of source models for our transformation directly from source code. This may also enable (partial) formalisation and automation of our normalisation step. Second, at the other end of the migration, we want to extend our approach with code generation from TRS models for the application-specific modules of the TRS product-line architecture, again using technologies related to the MDA. This would provide for a full-fledged model-driven migration approach: from legacy code to new code through a series of model transformations.

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, Dominique Perdaen, Joost Worms and Asia van de Mortel-Fronczak for their input and feedback. Finally, we thank Slinger Jansen of Utrecht University for his comments.

References

Baxter, I. D., Pidgeon, C., Mehlich, M., 2004. DMS: Program transformations for practical scalable software evolution. In: Proc. 26th Int'l Conf. Software Engineering (ICSE 2004). IEEE CS, pp. 625–634.

Bosch, J., Molin, P., 1999. Software architecture design: evaluation and transformation. In: Proc. 6th Symposium on Engineering of Computer-Based Systems (ECBS '99). IEEE CS, pp. 4–10.

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

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

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures:Views and Beyond. Addison-Wesley.

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

Fahmy, H., Holt, R. C., 2000a. Software architecture transformations. In: Proc. 16th Int'l Conf. Software Maintenance (ICSM 2000). IEEE CS, pp. 88–96.

Fahmy, H., Holt, R. C., 2000b. Using graph rewriting to specify software architectural transformations. In: Proc. 15th IEEE Int'l Conf. Automated Software Engineering. IEEE CS, pp. 187–196.

Gohari, P., Wonham, W., 2003. Reduced supervisors for timed discrete-event systems. IEEE Trans. Automatic Control 48 (7), 1187–1198.

Graaf, B., Lormans, M., Toetenel, H., November–December 2003. Embedded software engineering: state of the practice. IEEE Software 20 (6), 61–69.

Gray, J., Zhang, J., Roychoudhury, S., Wu, H., Sudarsan, R., Aniruddha, Neema, S., Shi, F., Bapty, T., 2004. Model-driven program transformation of a large avionics framework. In: Proc. 3rd Int'l Conf. Generative Programming and Component Engineering (GPCE 2004). Springer-Verlag, pp. 361–378.

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

Jouault, F., Kurtev, I., 2005. Transforming models with ATL. In: Proc. of the Model Transformations in Practice Workshop at MoDELS2005.

Lange, C. F., Chaudron, M. R., Muskuens, J., March 2006. In practice: UML software architecture and design description. IEEE Software 23 (2), 40–46.

Lehman, M. M., Belady, L. A. (Eds.), 1985. Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA.

MDR, 2006. mdr: netbeans.org : Metadata repository. `http://mdr.netbeans.org/`.

OMG, 2001. OMG Unified Modeling Language Specification, version 1.4. `http://www.uml.org`.

OMG, 2005a. MDA. `http://www.omg.org/mda`.

OMG, 2005b. Meta-object facility (MOF). `http://www.omg.org/mof`.

Perry, D. E., Wolf, A. L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40–52.

Ramadge, P., Wonham, W., 1987. Supervisory control of a class of discrete event processes. SIAM J. Control and Optimization 25 (1), 206–230.

Reveliotis, S. A., 2005. Real-Time Management of Resource Allocation Systems. A Discrete Event Systems Approach. Vol. 79 of Int'l Series in Operations Research & Management Science. Springer-Verlag.

Sabuncuoglu, I., Bayiz, M., 2000. Analysis of reactive scheduling problems in a job-shop environment. European J. operational research 126, 567–586.

Van den Nieuwelaar, N., 2004. Supervisory machine control by predictive-reactive scheduling. Ph.D. thesis, Technische Univ. Eindhoven.

Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C., 2004. Symphony: View-driven software architecture reconstruction. In: Proc. of the 4th Working IEEE/IFIP Conf. Software Architecture (WICSA 4). IEEE CS, pp. 122–134.

Viennot, G. X., 1986. Heaps of pieces, I: Basic definitions and combinatorial lemmas. In: Labelle, G., Leroux, P. (Eds.), Proc. Colloque de combinatoire énumérative (UQAM 1985), Montreal, Canada. Vol. 1234 of Lecture Notes in Math.. Springer-Verlag, pp. 321–350.