A MULTI-PARADIGM APPROACH FOR MODELLING SERVICE INTERACTIONS IN MODEL-DRIVEN ENGINEERING PROCESSES

Simon Van Mierlo[1] Yentl Van Tendeloo[1] István Dávid[1,2]
Bart Meyers[1,2] Addis Gebremichael[1] Hans Vangheluwe[1,2,3]

[1]University of Antwerp, Belgium
[2]Flanders Make vzw, Belgium
[3]McGill University, Montréal, Canada
{Simon.VanMierlo, Yentl.VanTendeloo, Istvan.David, Bart.Meyers, Hans.Vangheluwe}@uantwerp.be
Addis.Gebremichael@student.uantwerp.be

ABSTRACT

To tackle the growing complexity of engineered systems, Model-Driven Engineering (MDE) proposes to promote models to first-class citizens in the development process. Within MDE, Multi-Paradigm Modelling (MPM) advocates modelling every relevant aspect of a system explicitly, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, while explicitly modelling the underlying process. Often, activities of the process require interaction with (domain-specific) engineering and modelling tools. These interactions are, however, typically captured in scripts and program code, which is ill-suited for describing the timed, reactive, and concurrent behaviour of these protocols. Additionally, formal analysis of the overall process is limited due to the incorporation of black-box activities. In this paper, we propose an approach for the explicit modelling of service interaction protocols in the activities of MDE processes. We also explicitly model the execution semantics of our process model, to promote reuse and allow for future analysability. For both purposes, we propose to use SCCD, a Statecharts variant, resulting in a unified and concise formalism.

Keywords: process modelling, reactive systems, multi-paradigm modelling, service orchestration

1 INTRODUCTION

The complexity of engineered systems in increasing. This is mainly due to their heterogeneity, both at run time and design time. At run time, software controls hardware components in a feedback loop, the complete system has to interact (safely) with the environment, and often multiple such systems are connected over a network and have to communicate to achieve a task. At design time, these runtime requirements require multiple languages and tools to be combined to create a single big system. With the advent of Cyber-Physical Systems (CPS), smart mechatronic systems of the Industry4.0 initiative, and the Internet-of-Things (IoT), engineers are facing challenges of an unprecedented magnitude.

To successfully and efficiently tackle the complexity of the engineered system, modelling- and simulation-based techniques are increasingly used in the flow of the engineering work. *Model-Driven Engineering (MDE)* (Kent 2002) regards models as first-class concepts during system development: before realizing the system, stakeholders (*e.g.*, control engineers, application engineers, test engineers) build models of the various aspects (*e.g.*, physical, mechanical, software) of the system, resulting in a virtual product which can be analysed, simulated and verified. Stakeholder models capture the parts of the virtual product relevant to the

stakeholder (Broman et al. 2012). Often, stakeholders specialize in different domains and, therefore, their models are domain-specific. Within MDE, *Multi-Paradigm Modelling* (MPM) (Mosterman and Vangheluwe 2004) actively promotes this specialization. MPM advocates modeling every relevant aspect of the system explicitly, using the most appropriate formalism(s), at the most appropriate level(s) of abstraction, while explicitly modelling the process.

Such processes aim at depicting how the various domain-specific models are used during development. Models are passed around in the process and are being worked on within the activities of the process. These activities are either manual or automated, and typically make use of various services offered by engineering tools. If modelled in an appropriate formalism, the process can be analysed and subsequently enacted (Osterweil 1987). The enacted process orchestrates the engineering services, thus enabling a higher level of automation in the flow of the modelling work in general.

Orchestration requires a detailed specification of the interaction protocol with external services. In manual activities, user input is required, often through a (visual) modelling and simulation tool (for example, to create a model). In automated activities, a service (or multiple services) might be invoked and communicated with in an automated way (for example, to run a simulation). Such interaction protocols exhibit timed, reactive, and concurrent behaviour, making their formal analysis paramount in industrial-scale engineering processes. The analysis of the interaction protocols can improve its overall process with regards to transit time, scheduling, resource utilization, and overall model consistency. These interactions are, however, typically specified in scripts or program code, which interface with the API of the tools providing the services. Such an encoding of the interaction protocols inhibits their formal analysis.

Our contribution is twofold. First, we propose to explicitly model the external service interaction protocols in the activities of engineering processes using SCCD (Van Mierlo et al. 2016), a variant of Statecharts (Harel 1987). SCCD is appropriate for modelling timed, reactive, autonomous, and dynamic-structure behaviour, as it has native constructs available for it. This facilitates the implementation of the interaction protocols, and enables future analysis of the service orchestration. Second, we provide execution semantics for the overall process, expressed as an FTG+PM model, by mapping it to an SCCD model, augmented with process semantics. This avoids the need to define operational semantics for activity diagrams, which is non-trivial.

In the remainder of this section, we present a motivating example, used throughout the paper. Section 2 presents the FTG+PM and SCCD. Section 3 reviews related work to identify shortcomings in the state of the art. Section 4 presents the modelling of activities using SCCD. Section 5 presents the mapping of an FTG+PM to SCCD. Finally, Section 6 concludes the paper.

Motivating example

Our motivating example is the optimization of the number of traffic signals in a railway system. The system consists of sequences of railway segments, each guarded by a single traffic signal. For safety reasons, only one train is allowed per railway segment. Adding traffic signals increases the throughput of the system, though increases the cost of maintenance. The ideal number of traffic signals is therefore dependent on the characteristics of the system (*e.g.*, train inter-arrival time, acceleration, total length of the track).

The optimization is done by modelling the system with the DEVS formalism (Zeigler, Praehofer, and Kim 2000). Our problem requires several atomic DEVS models, such as a generator, collector, railway segment, and a traffic signal, and a single coupled DEVS model, coupling these atomic models together. This model is subsequently simulated for a fixed set of parameters, while varying the number of traffic signals over the total length. All simulation results are collected, the cost function is evaluated for all of them, and the number of traffic signals with the minimal cost is returned.

This process is shown in Figure 1, where we first design the various atomic models manually, though concurrently. As such, multiple engineers can model different aspects of the system concurrently. Additionally, a set of parameters is chosen, for which to simulate the model. Afterwards, the created atomic DEVS models are used to create the coupled DEVS model. It is this collection of models that is passed on to the optimization step, which plots out the costs of various configurations.

We implemented this example in the Modelverse (Van Tendeloo and Vangheluwe 2017, Van Tendeloo 2015), our prototype Multi-Paradigm Modelling tool. Simulations were performed using PythonPDEVS (Van Tendeloo and Vangheluwe 2014) as an external service.

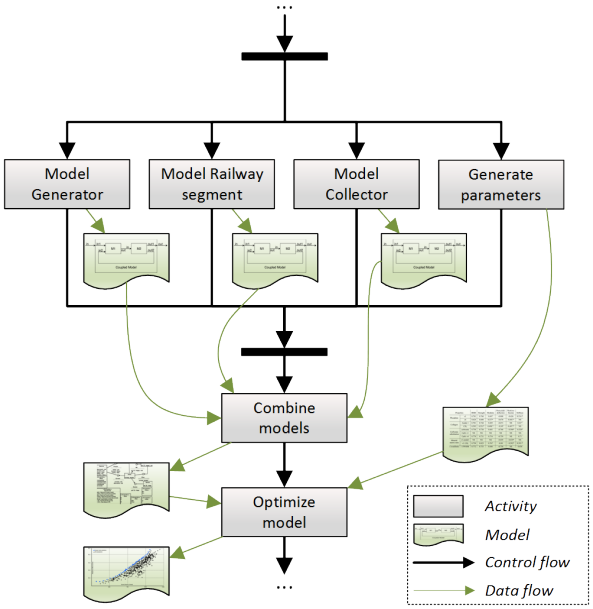

Figure 1: Process model of the example.

2 BACKGROUND

We first give a brief overview on the background of our work. Our approach builds on the combination of two formalisms: *Formalism Transformation Graph + Process Model* (FTG+PM) and *Statecharts + Class Diagrams* (SCCD), both of which are explained next.

2.1 Formalism Transformation Graph + Process Model (FTG+PM)

Process modelling is a widely used technique on the business level of a project. Business process modelling formalisms, however, fall short of capturing the essence of the engineering nature of complex system development. The *Formalism Transformation Graph + Process Model* (FTG+PM) formalism (Lucio et al. 2013) is designed specifically for depicting model-driven development processes. The two parts of the formalism depict the two aspects of an MDE process: the formalisms used throughout the process (FTG), and the process itself (PM). The FTG depicts the formalisms and the transformations between them, as used during the engineering process. The PM models the process and the models flowing through that process as artefacts. As such, the FTG serves as a type system for the PM, with formalisms typing artefacts and transformations typing activities. The PM allows to use any process modelling formalism, *UML Activity Diagrams* being the typical choice for this purpose (Dávid et al. 2017).

2.2 Statecharts + Class Diagrams (SCCD)

Statecharts is a formalism for modelling timed, reactive, autonomous systems, and was introduced by Harel (Harel 1987). Its main abstractions are states that can be composed hierarchically and orthogonally; transitions between these states that are either spontaneous, or triggered by an external event (coming from the environment), an internal event raised by an orthogonal component, or a timeout; and actions that are executed when a transition is executed.

While *Statecharts* is an appropriate formalism for describing the timed, reactive, autonomous behaviour of systems, it does not allow to model a system with dynamically changing structure. In many systems, objects are continuously created and destroyed. The SCCD formalism (Van Mierlo et al. 2016) extends the *Statecharts* formalism with the concepts of the *Class Diagrams* formalism (classes and relations, which

model structure). Each class in the class diagram is associated with a definition of its behaviour (in the form of a *Statecharts* model). At runtime, an object can request for a class to be instantiated as an object, and relationships between classes to be instantiated as links between objects. Links serve as communication channels, over which objects can send and receive events. There is exactly one default class, of which an instance is created when the system is started by the runtime.

3 RELATED WORK

Process and workflow modelling is an extensively researched domain. Process modelling languages are primarily geared towards modelling concurrency and synchronisation (van der Aalst et al. 2003). Pertinent examples include languages based on the *Business Process Modelling Notation* (Stiehl 2014), *Petri nets* (van der Aalst 2015) and *UML Activity Diagrams* (Bhattacharjee and Shyamasundar 2009). We focus on and discuss the most relevant and well-known approaches in terms of the intentions of this paper next.

The *Business Process Modelling Notation* (BPMN) (Silver and Richard 2009) is a widely used standard in process modelling. BPMN is used in a wide range of areas, to model processes in non-IT, as well as IT-intensive organisations. Its main goal is to provide an understandable notation for all stakeholders. The focus is more on the conceptual modelling of processes, and less on orchestration and execution. In version 2.0, the standard has been extended with support for orchestration, albeit on a non-technical level.

jBPM (Cumberlidge 2007) is an open-source, Java-based framework that supports execution of BPMN 2.0 conform processes. The framework also provides enhanced integration features with external services in the form of managed Java program snippets. In addition, the process engine is tightly integrated with a collaboration and management service (Guvnor), a standardized human-task interface (WS-HT), a rule engine (Drools) and a complex event processing engine (Drools Fusion).

The *Business Process Execution Language* (BPEL) (Weerawarana et al. 2005) is a standardised language for specifying activities by means of web services. The standard specifies a BPEL process as XML code, though graphical notations exist, often based on BPMN. Service interaction can be executable or left abstract. Analysis tools for BPEL have been developed, for example by formalising BPEL models in terms of *Petri nets* as done by Ouyang et al. (2007) and Xia et al. (2012). Kovács, Varró, and Gönczy (2008) use a symbolic analysis model checker. Fu, Bultan, and Su (2004) and Foster et al. (2003) analyse the communication between BPEL processes by employing automata. Nevertheless, BPEL is exclusively used for web services defined using WSDL.

Open Services for Life-cycle Collaboration (OSLC) (OSLC Community 2017) is the de facto standard in tool integration. It is a specification for the management of software lifecycle models and data, which are represented as resources. The specification is intended to be used for integration of services and data, and does not include process modelling.

The *Statecharts* formalism (Harel 1987) has first-class notions of concurrency, hierarchy, time and communication. It can therefore be viewed as a suitable formalism for integration and orchestration. Because *Statecharts* is state-based, and does not include *fork* and *join* constructs, it is less suitable for process modelling. *Statecharts* has been combined with *Class Diagrams* in SCCD (Van Mierlo et al. 2016), to provide structural object-oriented language constructs (*i.e.*, objects with behaviour).

Story diagrams (von Detten et al. 2012) are a formal behavioral specification language with workflow semantics. Similarly to UML Activity Diagrams, they describe control and data flow across the workflow, but with the added support for specifying executable actions. Just like the FTG+PM formalism, story diagrams rely on typed attributed graph transformations, but with a very simplistic type model. As a result, event though story diagrams provide added behavioral specification semantics, the formalism still is not as versatile as the FTG+PM.

A summary of all approaches and their suitability for our purpose is presented in Table 1. We have investigated whether the approach is intended to be used to specify processes (**process**), whether it aims at integration of services/tools (**integration**), whether it supports execution or enactment (**executability**), whether it provides means for formal analysis (**analyzability**), and whether its notation is appropriate for the tasks it is intended for (**usability**).

Approach	Process	Integration	Executability	Analysability	Usability
Petri nets	●	○	●	●	◐
Activity Diagrams	●	○	●	●	●
BPMN2.0	●	○	●	●	●
jBPM	●	●	●	○	●
BPEL	●	◐	●	●	◐
OSGi	○	◐	●	○	●
OSLC	○	●	●	○	●
FTG+PM	●	○	●	●	●
SCCD	○	●	●	●	●
Story diagrams	○	●	●	◐	◐

Table 1: Summary of related work. (● - Supports, ◐ - Partially supports, ○ - Does not support)

The main conclusion is that no approach truly unifies process modelling and integration of services. The approaches that score best in these two aspects (BPEL and Orc) do not have an intuitive, accessible notation, although in the case of BPEL, graphical notations have been suggested but are not part of the standard. jBPM overcomes these shortcomings, but does not support analysability of the service interactions. In the remainder of this paper, we present an approach that scores well in all of the above aspects, by combining the FTG+PM and the SCCD formalisms.

4 MODELLING ACTIVITIES USING SCCD

We first turn to the definition of an activity. Activities are the atomic actions being executed throughout the enactment of the process. Up to now, we were agnostic of what is the content of the activity, as we merely require it to be executable. Most often, it is hardcoded in some programming language or provided as an executable binary. When control is passed to a specific activity, the activity executes.

4.1 Problem Statement

Activities can be hardcoded, but code is arguably not the optimal formalism to describe an activity. While activities can be limited to executing some local computation, it frequently requires external tool interaction. Such external tools can be anything, for example a (highly-optimized) simulator, or a modelling tool. In such cases, hardcoding the potentially complex interaction protocol is far from ideal. Indeed, the behaviour of protocols exhibits timing (*e.g.*, network timeouts, delays), reactivity (*e.g.*, responding to an incoming message), and concurrency (*e.g.*, orchestrating multiple tools concurrently).

In our running example, we see this exact problem occuring in the *"optimize model"* activity. In this activity, we want to optimize the cost for a given set of parameters, varying a single parameter within a given range. Concretely, we want to vary the number of traffic lights in the simulation, while keeping all other parameters fixed. In the end, the activity needs to return the optimal solution; that is, it returns the optimal number of traffic lights for the given set of parameters. In essence, the same simulation is ran with slightly different parameters. This is, however, embarassingly parallel: each simulation run is independent of every other simulation run. Therefore, we desire to run some simulations in parallel. Doing this the usual way (*i.e.*, with

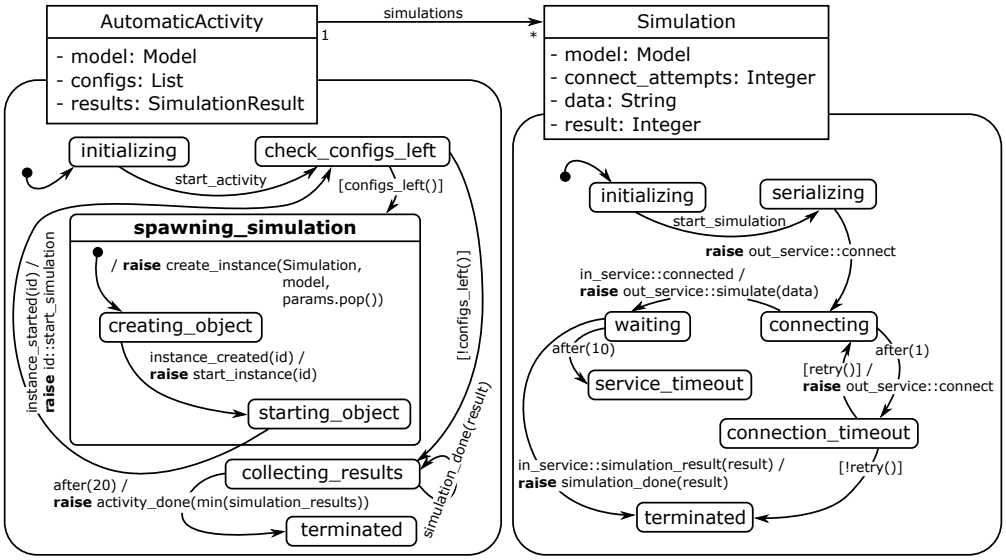

Figure 2: Automatic activity: protocol implemented to communicate with an external service.

code) is non-trivial: concurrency requires threads (which is problematic (Lee 2006)), reactivity requires the use of a main loop (possibly with polling), and timeouts require interruptable sleep calls.

4.2 Approach

The previous discussion illustrated that code is not ideal to specify timed, reactive, and concurrent activities, which is the case with service orchestration. We propose to use a formalism equipped with better support for these requirements: SCCD. Some activities are therefore ideally modelled with SCCD, where they can automatically make use of its features. On the implementation side, SCCD manages all concurrency, timing, and reactivity natively. Indeed, concurrency is supported by orthogonal components and dynamic structure, reactivity is supported by event-based transitions, and timeouts are supported by *after* events.

In our running example, we see that these features of SCCD are all required in the *"optimize model"* activity. Concurrency is required to spawn several instances of the simulator concurrently, and the number is only known at runtime, as it depends on the number of possible configurations. Reactivity is required to handle the results of these individual simulators, which should be aggregated. Timeouts are required to handle network timeouts and potential infinite simulations.

In Figure 2, we show how the example activity is modelled with SCCD. Thanks to SCCD, we can spawn an arbitrary number of *"Simulation"* objects, by sending out an event to the object manager, thereby allowing for dynamic structure (implementing *concurrent*). After a simulation is spawned dynamically, for each configuration to evaluate, we wait for results to come in, encoded in events (implementing *reactive behaviour*). Each of the spawned simulations serializes the model, and sends it to the actual external simulator, after which the simulator is started externally. If no response is received from the simulator during initialization before a timeout occurs, we retry the connection (implementing *timing*). If the simulation was started successfully, but no result comes in before a timeout occurs, we determine that the simulation has crashed, is stuck in an infinite loop, or ran out of memory. Independent of the reason, we determine that the simulation result is not the optimum, and subsequently ignore the simulation run. When all simulation results are in, or we have waited sufficiently long, we return the optimal parameter that we found.

5 MAPPING PROCESSES TO SCCD

Orthogonal to the previous section, where we modelled the contents of the activities using SCCD, we now look at the process model itself. The process model chains the different activities, dictating the order in which they should be executed, possibly concurrently. Of specific interest is the fork/join operation, which executes multiple activities concurrently and synchronizes when both have finished. This is ideal for manual activities, for which multiple developers might be involved, who can now model concurrently.

5.1 Problem Statement

Despite the advantages of concurrent manual activities, implementing this in a truly parallel fashion is non-trivial. Basic implementations merely dictate an arbitrary order between different concurrent activities, without actually executing them in parallel. This was originally the case in our prototype tool, the Modelverse, because true concurrency is difficult and relies on many platform characteristics. Examples are the choice between processes or threads, their interleavings, how the implementation platform supports parallelism, and how data is shared between activities. These are only a small selection of crucial questions regarding the implementation of process enactment. A significant investment to implement and maintain this infrastructure is needed if processes are implemented using traditional (code-based) techniques.

For our running example, this is shown in the concurrent manual activities in the beginning of the process: creating the various DEVS models. These models are independent, and can easily be created in parallel. Nonetheless, if there is no support for activities to run concurrently, all work is effectively sequentialised, significantly increasing the duration of the overall process.

5.2 Approach

The problem arises due to the lacking native support for concurrency in many implementation languages. As such, implementing process model enactment requires many workarounds to achieve true parallelism. We note, however, that languages do exist that natively support notions of concurrency, for example SCCD. Nonetheless, as mentioned in section 3, SCCD was not designed to model workflows, and is therefore not suited for direct modelling. In summary, we want users to model using activity diagrams, as they are used to, but for execution purposes, we transform the modelled process to an SCCD model. This transformation defines denotational semantics for process models, instead of operational semantics (an executor).

While other such languages exist, we favour SCCD as this allows us to reuse the SCCD execution engine, needed to execute activities. Additionally, we see many future opportunities for our approach if both orthogonal dimensions (modelling activities with SCCD, and mapping the process to SCCD) are combined: both share the same (hierarchical) formalism, and can therefore potentially be flattened.

Mapping activity diagrams to SCCD can be achieved through the use of model transformations, which are often referred to as the heart and soul of MDE (Sendall and Kozaczynski 2003). With model transformations, a Left-Hand Side (LHS) is searched throughout the model, and, when matched, the match is replaced with a Right-Hand Side (RHS), if the Negative Application Condition (NAC) does not match at the same time. In our case, the LHS consists of activity diagrams elements, such as the *activity* construct, while the RHS copies the activity diagram construct (thereby leaving the activity diagram intact) and creates an equivalent SCCD construct (*i.e.*, an orthogonal component). Defining such a mapping is significantly less work than defining operational semantics from scratch, as we will show. Additionally, by mapping to SCCD, there is only one implementation of an executor for timed, reactive, autonomous, dynamic-structure behaviour that must be maintained (the SCCD executor).

A naive mapping to SCCD would map forked activities to orthogonal components, each spawning and managing the execution of the activities; joins synchronize the execution by transitioning from the end states of these components. While intuitive, this mapping runs into problems with non-trivial concurrent regions. For example, consider two parallel forks that interleave: the two forks cannot be independently mapped, as they interact with one another, resulting in a different mapping to orthogonal regions.

We define a more generic mapping, based on orthogonal components, one for each activity diagrams construct. The order in which the orthogonal components are enabled, is defined by the condition that is present in the orthogonal component itself. Each orthogonal component checks whether it has the "execution token", and if so, it passes on the token. All orthogonal components execute concurrently, meaning that if multiple tokens exist, multiple orthogonal components operate concurrently. Depending on the type of construct, the behaviour changes: activities execute and pass on the token upon completion, a fork splits the token, a join merges tokens, and a decision passes the token conditionally. In the remainder of this section, we describe our transformation rules for each activity diagram construct in detail.

5.3 Transformation Rules

The following transformation rules are executed in the presented order. Before we actually start the translation, however, we first perform a minor optimization step: subsequent fork operations are merged into a single fork. This is not performed for performance considerations, but makes the mapping slightly easier. This optimization thereby removes subsequent forks, allowing us to skip this case in the remainder of the mapping. The same happens for join nodes.

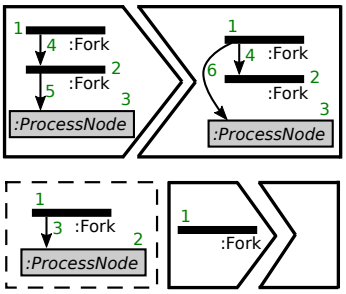

Figure 3: Optimize rules.

Optimization Figure 3 presents the optimization of fork nodes. The first (topmost) rule makes sure that the first fork directly links to all targets of the second fork, removing the target from the second fork. This rule keeps the model semantically equivalent, as the second fork now has no successors. In the second (bottommost) rule, an empty fork node is removed, as it has no outgoing edges any more. This rule again maintains semantic equivalence, as the second fork has no successors left. Similar rules exist for the optimization of fork nodes.

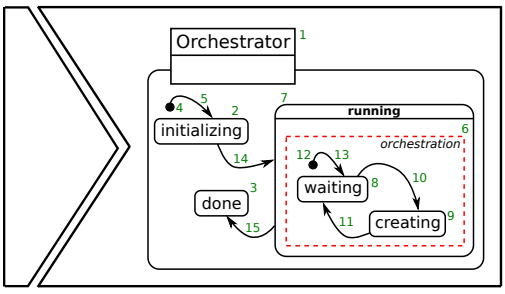

Figure 4: Orchestrator rule.

Orchestrator Figure 4 presents the transformation rule for the orchestrator, which executes once. Each subsequent transformation rule extends a single composite state with an orthogonal region. The orthogonal regions execute all elements of the activity diagram in parallel, waiting for a condition to become true. The first step consists of creating the composite state and providing it with an orthogonal region that catches a spawn event, and performs the spawning of an activity. By defining this code here, it does not have to be reproduced throughout the other orthogonal regions, maximizing reuse.

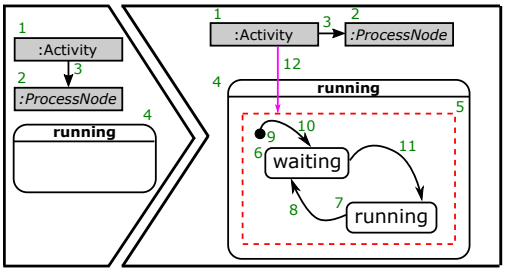

Figure 5: Activity rule.

Activity Figure 5 presents the transformation rule that executes for each activity. Activities are relatively easy to map, as they merely require the spawning of their associated activity (which, in our case, is modelled by another SCCD class). This is achieved by sending a spawn event to the orchestrator, and transitioning to a "running" state. We stay in this state until we have determined that the spawned activity has terminated, after which we mark the current activity as executed (*i.e.*, we pass on the token).

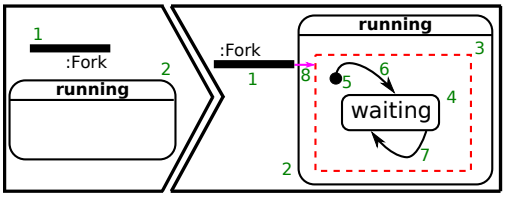

Figure 6: Fork rule.

Fork Figure 6 presents the transformation rule that executes for each fork node. Forking requires a single token to be distributed among all of its successors, without doing any computation itself. As such, our transformation rule adds an orthogonal component which continuously polls whether or not it has received the token. If it receives the token, it immediately passes the token to all of it successors simultaneously.

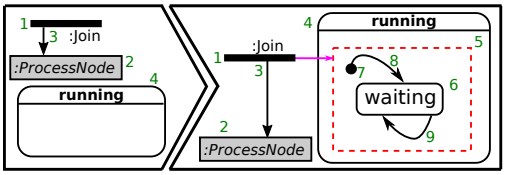

Figure 7: Join rule.

Join Figure 7 presents the transformation rule that executes for each join node. Joining is slightly more complex: it has to check for multiple tokens, before becoming enabled. When enabled, it consumes all of these tokens and passes on the token to its own successor, of which there is only one.

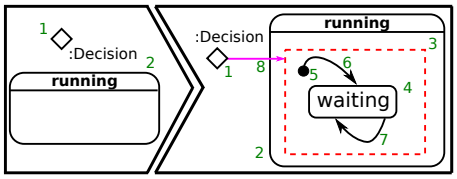

Figure 8: Decision rule.

Decision Figure 8 presents the transformation rule that executes for each decision node. The final construct that we have to map, is the decision node. Similar to all previous nodes, we check whether we have a token to start execution. Depending on the input data that we receive, we decide to pass on the token to either the *true*- or the *false*-branch.

5.4 Result

The resulting mapped model for our motivating example is shown in Figure 9. The Orchestrator class is the default class of the SCCD and spawns the other classes (i.e., the activities) depending on its orthogonal components. These other classes are, for example, the optimization activity previously described in Figure 2. Other classes are not shown due to space restrictions.

6 CONCLUSION AND FUTURE WORK

In the context of MPM, service orchestration is essential for the combination of multiple external tools. Nonetheless, current approaches do not sufficiently address the challenges it poses: timed, reactive, and concurrent behaviour. In this paper, we propose an approach for handling these problems by two contributions, based on SCCD (a *Statecharts* variant), which has native notions of timing, reactivity, concurrency,

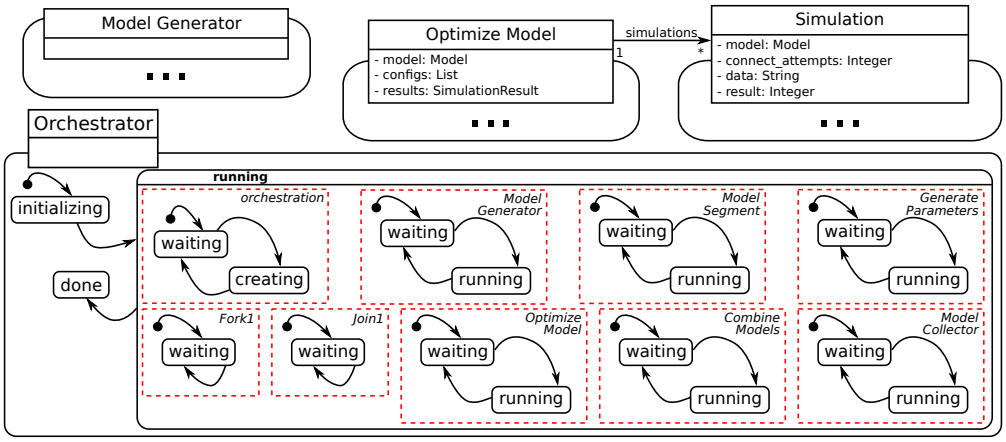

Figure 9: Process model from Figure 1 mapped to SCCD.

and dynamic structure. First, activities themselves are modelled using SCCD, allowing external service protocols to be more effectively specified. Second, the process model is transformed into an equivalent SCCD model for execution. This preserves the modelling abstractions provided by activity diagrams, while gaining the execution of SCCD.

In future work, we plan to consider the benefits of combining these two orthogonal dimensions of our approach. Indeed, as both the process and activities are modelled in SCCD, they can be combined into a single SCCD model. This single SCCD model can subsequently be analysed (Pap et al. 2005) or debugged (Mustafiz and Vangheluwe 2013), without any additional work. To achieve the valid and sound construction of this combined SCCD interaction/process model, composition rules of the single interaction SCCD model need to be investigated. Our previous work on process-oriented inconsistency management in MPM settings (Dávid et al. 2016) is a prime candidate to be augmented with such an approach. *Software Process Improvement* (SPI) techniques in general can greatly benefit from our approach as well.

REFERENCES

Bhattacharjee, A. K., and R. K. Shyamasundar. 2009. "Activity Diagrams : A Formal Framework to Model Business Processes and Code Generation". *Journal of Object Technology* vol. 8 (1), pp. 189–220.

Broman, D., E. A. Lee, S. Tripakis, and M. Törngren. 2012. "Viewpoints, formalisms, languages, and tools for cyber-physical systems". In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pp. 49–54. ACM.

Cumberlidge, M. 2007. *Business process management with JBoss jBPM*. Packt Publishing Ltd.

Dávid, I., J. Denil, K. Gadeyne, and H. Vangheluwe. 2016. "Engineering Process Transformation to Manage (In)consistency". In *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016)*, pp. 7–16. http://ceur-ws.org/Vol-1717/.

Dávid, I., B. Meyers, K. Vanherpen, Y. Van Tendeloo, K. Berx, and H. Vangheluwe. 2017. "Modeling and Enactment Support for Early Detection of Inconsistencies in Engineering Processes". In *2nd International Workshop on Collaborative Modelling in MDE*.

Foster, H., S. Uchitel, J. Magee, and J. Kramer. 2003. "Model-based Verification of Web Service Compositions". In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pp. 152–163.

Fu, X., T. Bultan, and J. Su. 2004. "Analysis of interacting BPEL web services". In *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pp. 621–630.

Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems". *Sci. Comput. Program.* vol. 8 (3), pp. 231–274.

Kent, S. 2002. *Model Driven Engineering*, pp. 286–298. Berlin, Heidelberg, Springer Berlin Heidelberg.

Kovács, M., D. Varró, and L. Gönczy. 2008. "Formal analysis of BPEL. workflows with compensation by model checking". *Comput. Syst. Sci. Eng.* vol. 23 (5).

Lee, E. 2006. "The Problem with Threads". Technical report, University of California at Berkeley.

Lucio, L., S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. 2013. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, pp. 182–202.

Mosterman, P. J., and H. Vangheluwe. 2004, September. "Computer Automated Multi-Paradigm Modeling: An Introduction". *Simulation* vol. 80 (9), pp. 433–450.

Mustafiz, S., and H. Vangheluwe. 2013. "Explicit Modelling of Statechart Simulation Environments". Proceedings of the Summer Simulation Multiconference, pp. 21:1–21:8.

OSLC Community 2017. "OSLC - Open Services for Lifecycle Collaboration Core Specification Version 3.0". http://open-services.net.

Osterweil, L. 1987. "Software Processes Are Software Too". In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pp. 2–13, IEEE Computer Society Press.

Ouyang, C., E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. 2007. "Formal semantics and analysis of control flow in WS-BPEL". *Sci. Comput. Program.* vol. 67 (2-3), pp. 162–198.

Pap, Z., I. Majzik, A. Pataricza, and A. Szegi. 2005. "Methods of Checking General Safety Criteria in UML Statechart Specifications". *RELIABILITY ENGINEERING & SYSTEM SAFETY* vol. 87, pp. 89 – 107.

Sendall, S., and W. Kozaczynski. 2003. "Model Transformation: The Heart and Soul of Model-Driven Software Development". *IEEE Softw.* vol. 20 (5), pp. 42–45.

Silver, B., and B. Richard. 2009. *BPMN method and style*, Volume 2. Cody-Cassidy Press Aptos.

Stiehl, V. 2014. *Process-Driven Applications with BPMN*. Springer.

van der Aalst, W. 2015. "Business process management as the "Killer App" for Petri nets". *Software and System Modeling* vol. 14 (2), pp. 685–691.

van der Aalst, W., A. ter Hofstede, B. Kiepuszewski, and A. Barros. 2003. "Workflow Patterns". *Distributed and Parallel Databases* vol. 14 (1), pp. 5–51.

Van Mierlo, S., Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. 2016. "SCCD: SCXML Extended with Class Diagrams". In *3rd Workshop on Engineering Interactive Systems with SCXML*.

Van Tendeloo, Y. 2015. "Foundations of a Multi-Paradigm Modelling Tool". In *MoDELS ACM Student Research Competition*, pp. 52–57.

Van Tendeloo, Y., and H. Vangheluwe. 2014. "The Modular Architecture of the Python(P)DEVS Simulation Kernel". In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS*, pp. 97–102.

Van Tendeloo, Y., and H. Vangheluwe. 2017. "The Modelverse: a Tool for Multi-Paradigm Modelling and Simulation". In *Proceedings of the Winter Simulation Conference*, pp. 944–955.

von Detten, M., C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. 2012. "Story diagrams-syntax and semantics". *Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Tech. Rep. tr-ri-12-324*.

Weerawarana, S., F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. 2005. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR.

Xia, Y., Y. Liu, J. Liu, and Q. Zhu. 2012. "Modeling and Performance Evaluation of BPEL Processes: A Stochastic-Petri-Net-Based Approach". *IEEE Trans. Systems, Man, and Cybernetics, Part A* vol. 42 (2), pp. 503–510.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Academic Press.

ACKNOWLEDGEMENTS

This work was partly funded by PhD fellowships from the Research Foundation - Flanders (FWO) and Agency for Innovation by Science and Technology in Flanders (IWT); by Flanders Make vzw, the strategic research centre for the manufacturing industry; and by the Flanders Make project MBSE4Mechatronics (grant nr. 130013) of the IWT.

AUTHOR BIOGRAPHIES

SIMON VAN MIERLO is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). For his PhD, he is studying how modelling formalisms, environments, and simulators can be enhanced with debugging support.

YENTL VAN TENDELOO is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). In his Master's thesis, he worked on MDSL's PythonPDEVS simulator, a simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS, grafted on the Python programming language. The topic of his PhD is the conceptualization, development, and distributed realization of a new (meta-)modelling framework and model management system called the Modelverse.

ISTVÁN DÁVID is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). He obtained his Master's degrees from the Budapest University of Technology and Economics. His research interests include process modelling, inconsistency management in multi-model/multi-paradigm settings, complex event processing and language engineering.

BART MEYERS is a post-doc in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). His research interests are model-driven engineering, domain-specific modelling and language engineering.

ADDIS GEBREMICHAEL holds a Master's degree in Computer Science from the University of Antwerp.

HANS VANGHELUWE is a Professor in the department of Mathematics and Computer Science at the University of Antwerp (Belgium) and an Adjunct Professor in the School of Computer Science at McGill University (Canada). He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results.