Solution Workflows for Model Based Analysis of Complex Systems

Francesco Moscato, Valeria Vittorini, Flora Amato, Antonino Mazzeo, Nicola Mazzocca

Abstract—The development and analysis of increasingly complex systems require the intensive use of models and of sophisticated approaches to systems modeling. This paper focuses on workflows supporting the solution of complex, composed, formal models used to study and/or develop real-world systems. The workflows we deal with orchestrate multiple distributed tools and applications in order to provide the user with a powerful, composed solution environment. The aim is to automate and reproduce analysis and simulation tasks starting from a high level, graph-based description of the model to be solved. The paper thus introduces solution workflows and presents the Solution Process Definition Language (SPDL) for the specification of solution workflows processes. One of the key elements of SPDL is its formal semantics, which allow for unambiguous specification of its constructs and validation of the workflows. A workflow pattern analysis of SPDL is also provided. SPDL and its execution environment, the OsMoSys framework, are then applied to a homeland security scenario. The OsMosys framework and the SPDL language provide a practical contribution to the applicability of model engineering techniques by enabling the semi-automatic solution of complex models.

Note to Practitioners - The usage of formal methods is widely advocated to provide evidence of critical properties of systems for certification and/or assessment purposes. This paper was motivated by the problem of (partially) automating the development and the analysis of complex formal models of real systems. Existing approaches generally require that whoever builds such models has an in-depth knowledge of several fields, besides the domain to study: modeling languages (e.g. Petri Nets, Fault Trees, Process Algebras, Queuing Networks, Bayesian Networks), solution techniques, analysis/simulation tools, as well as the way to combine them in order to perform more powerful experiments. In this paper we propose a workflow based solution, in order to enable the automated development and analysis of multi-formalism models by composing pre-defined models and existing tools. Our approach also promotes the reuse of models and experiments, through the construction of libraries of models and workflows and the definition of sophisticated experiments.

Index Terms—Complex Systems, Formal Modeling, Model Engineering, Scientific Workflow, Workflow Patterns.

I. INTRODUCTION

Developing increasingly complex systems requires proper methodologies and tools in order to meet requirements and reduce system development costs and time. Model-based development methods and supporting technologies may cope with this challenge but they often require the building and analysis of very complex models. These models are possibly developed according to compositional and multi-formalism approaches and consist of several sub-models (expressed by means of different modeling languages). The work presented in this paper is motivated by the problem of supporting and (partially) automating the analysis of complex formal models of real systems, thus enabling the practical usage of model engineering techniques. The automation of the process that must be enforced to solve a complex model is a particular kind of workflow, here called solution workflow. Solution workflows offer several advantages in solving and analyzing complex models: they greatly improve the efficiency of the solution processes and support multiple experiments, allow for reuse of process definitions, promote the development of flexible and dynamic models, enable the reuse and the sharing of existing models, and the use of external (available) computational resources. The aim of the paper is to introduce solution workflows and to present the Solution Process Definition Language (SPDL). The paper also addresses the semi-automatic generation of SPDL solution workflows starting from a graph based description of the model to be solved. Important features of SPDL and of its solution framework OsMoSys, which justify the introduction of an ad-hoc workflow language, are: (a) the possibility of using the available solution and analysis tools without requiring any modifications; (b) the semi-automatic generation and execution of SPDL specifications from a high level description of the composed model; (c) the formal semantics on which SPDL is based, allowing a rigorous definition of its constructs and the formal verification of the correctness of the flow; (d) the expressive power of SPDL, in terms of the workflow patterns that it is able to implement.

Below we provide a small illustrative example that clearly states the problem the paper deals with and places our work in the context of both multi-formalism modeling and workflow management fields.

A. Problem Statement

Consider a GRID system where several computing nodes (CNs) and storage nodes (SNs) interact to execute a parallel application. A Scheduler chooses the resources to reserve for the elaboration according to performance and availability measures provided by a performance and availability model of the system.

A simple model of the system may consist of the models of the single CN and SN nodes connected to each other, where each CN/SN model in turn consists of two components: a performance sub-model and an availability sub-model which
are composed to take into account the dependence existing between the performance level and the availability level of the node. In this example, low performance affects the node availability. The sub-models may be expressed by using different modeling formalisms: the performance models could be defined by using Petri Nets for the CN nodes and Queuing Networks for the SN nodes, the availability models may be Fault Trees, and so on. Models may also be replicated to describe different CN (SN) nodes: different values of proper parameters may be used to differentiate the nodes (i.e. CPU speed, disk throughput etc.). The connections between (sub-)models represent the way in which the information produced during the analysis of the models must be used. Once the entire composed model of the system has been built, the user wants to analyze and solve it. This requires that several steps are performed in the right order, from indices definition to the solution of the sub-models, to the collection of the intermediate results provided by the performance sub-models, to the usage of these intermediate results to perform the analysis of the availability sub-models. These steps involve the execution of several analysis tools and require data routing among them. In conclusion, the analysis of the entire model is a complex process that is hard to define and is executed by hand. This kind of process, which we call a solution process, can be effectively defined by means of workflow languages and its specification and execution can be automated.

The automated generation of solution processes is appealing for users who want to define the whole model by composing sub-models from libraries, but several problems arise when dealing with existing workflow languages. First of all, no workflow language has been defined with the aim of managing models, defining measures and indices to be evaluated and handling the results produced by different solvers and analysis tools. Furthermore, the automated generation of the solution process needs a formal definition of the semantics of the workflow language used to specify the solution processes. This is a problem, since the available workflow languages lack formal semantics. Finally, the expressive power of the workflow language - and consequently the possibility of specifying several workflow patterns - is an important issue in this context, because the analysis tools can be required to interact according different specific patterns in order to solve the model.

B. Related Literature

The previous example shows how the related literature relevant to our work ranges from multi-formalism modeling to workflow management. Here we briefly review the available results in the field of the development of multi-formalism models and analysis, and provide a discussion on workflow based approaches and languages which are closer to the main focus of the present paper.

One of the first tools in the multi-formalism field is SHARPE [1]–[3] which focuses on performance and dependability evaluation. SHARPE allows for modular composition of models and supports both multiple solvers and multi-formalism. In SHARPE the solving process is somehow guided by the modeler, interactively or via batch files. To the best of our knowledge, SHARPE does not natively support the extension of the number of formalisms or solvers it can use, nor needs a dedicated subsystem for results handling and presentation.

The DEDS toolbox [4] supports multi-formalism and multiple solvers as well. One of the main features of the DEDS toolbox is its design for extension. This is obtained by the introduction of a high level (text-based) interface format (namely APNN, Abstract Petri Nets Notation), shared by all the components of the toolbox architecture.

The MODEST approach [5], [6] (supported by the tool-suite MOTOR) is based on a single formalism for systems specification that is used to write models solved by multiple solvers. Hence, the language (that is an extended process algebra) has a non-trivial intrinsic complexity and each solver extracts a simpler model containing the needed information from the main model.

Möbius [7]–[9] supports both multi-formalism and multiple interacting solvers. Models can be composed in order to exploit modularity and multi-formalism, obtaining a composed model.

Finally, the definition and execution of solution processes proposed by this paper is supported by the OsMoSys framework [10] which will be described in Section II in order to introduce some concepts used throughout the paper. Unlike the above mentioned frameworks, which are based on “integrative” approaches, OsMoSys is based on workflow and orchestration. This is very advantageous in terms of flexibility and reuse.

Scientific workflows are now emerging as an effective solution to enable complex analysis and computation in e-science.

Scientific workflow systems have been developed to enable GRID based approaches to large-scale analysis on scientific data (e.g. Taverna,1 The Gridbus Workflow,2 Kepler [11]).

The solution workflows we deal with can be considered a particular type of scientific workflows. They have some features in common with web service based workflows and specifically with service composition processes. Some contact points exist between SPDL, WS-BPEL [12] (BPEL for short) and XPDL [13]. A comparison between SPDL, BPEL and XPDL is given in Section III-C. The main problem with BPEL and XPDL (and similarly with other workflow languages), is that they lack formal semantic definitions. In particular, a study of XPDL and BPEL languages (in [14]), has shown that the XPDL document definition does not unambiguously specify all the elements in the language. As a result, many vendors can claim to be compliant with the language specifications while interpreting the same constructs in different ways.

The same holds for the BPEL language: in [15] it is also shown that BPEL comes without formal semantics and its specification document, written in “natural” language, contains a fair number of acknowledged ambiguous aspects that may lead to different interpretations. As result, several formal

1http://taverna.sourceforge.net/
2http://www.gridbus.org/workflow/
semantics of WS-BPEL were proposed in the literature (for an overview see [16]) but none of them is widely recognized and used.

C. Contribution and Outline

In this paper we provide two main contributions. We introduce the concept of solution workflow in order to effectively support the development and the analysis of multi-formalism models of complex systems, and propose the Solution Process Definition Language (SPDL) and its formal semantics to specify (and enact) solution workflows.

This work is placed within the context of the OsMoSys project which provides the methodological and practical basis for the development and the solution of multi-formalism models according to the concepts and the language introduced in the paper.

Moreover, in this paper we also briefly address the issue of the automated generation of SPDL specifications, since this is an important aspect to be taken into account in evaluating the real applicability of the results of our work.

The rest of this paper is organized as follows. In Section II, we describe the OsMoSys Project, the underlying modeling methodology and the architecture of the software framework which allows for the definition and the enactment of the solution workflows defined by using the SPDL language.

The SPDL language is introduced in Section III. To this end some basic notations are given in Section III-A. The operational semantics of SPDL are then defined in Section III-B, a comparison between SPDL and some reference workflow languages is undertaken in Section III-C by means of a detailed patterns analysis, and the specific features of SPDL related to the management of (formal) models are described in Section III-D.

Section IV briefly discusses some mechanisms for the automated generation of solution workflows. The main data-flow operators used to compile SPDL code starting from a graph based representation of the workflows are defined.

In Section V an application of SPDL and OsMoSys to a homeland security scenario is described.

Finally, Section VI summarizes the contributions of this paper and proposes a few directions for further research.

II. THE OSMO SYS PROJECT

The vision underlying the OsMoSys (Object-baSed multi-formalism M0deling of SYStems) project is to apply the concepts of component software engineering to the development of formal models [17], [18] in order to provide a practical support to model engineering.

Our work in this direction evolved from an initial notion of model composition based on pre-defined building blocks [19] to the development of a more sophisticated modeling approach [20] and the application of workflow concepts to the development and the analysis of formal models [21], [22]. The availability of a supporting methodology and tools allowed us to define extensions to existing formalisms in order to enhance their expressive power [23] and cope with the analysis of complex systems, for example in [24], [25], [26], [27].

In this Section we summarize the modeling methodology and describe the execution environment of the SPDL processes.

A. Modeling Methodology

The OsMoSys Modeling Methodology (OMM) provides a conceptual framework based on object orientation concepts and meta-modeling [28]–[30]. Specifically, OMM uses meta-modeling to achieve flexibility in adding and/or extending modeling languages, and in achieving formalisms interoperability when composing models expressed by different formalisms.

According to OMM, modeling languages are classes and new formalisms can be easily introduced or defined by inheritance from existing ones. Models are classes too, whose names identify the entities they represent. Every model instantiates the formalism element types to construct the model itself and the definition of an element type may specify several properties and/or results. For example a Petri Net model will instantiate the element types Places, Transitions and Arcs from the Petri Net formalism, while the property Marking and the result Mean are usually associated with the element type Place of a Generalized Stochastic Petri Net.

OMM models can be structured by including sub-models in other models, with the facilities typical of an object based environment. Complex models - called composed models - can be built by connecting instances of sub-models by proper composition operators through their interfaces. The interface of a model is the subset of elements used as source and destination operands of the composition operators. Since multi-formalism models consist of composed sub-models expressed by different modeling languages, the expressive power of each language is preserved and the features of native solution techniques and tools are available.

A multi-formalism model can be represented by a bipartite graph whose nodes are sub-models and operators. The arcs of the graph connect sub-models to operators and vice-versa. This graph-based representation of the model is called bridge model.

Fig. 1 shows a simple bridge model consisting of two sub-model instances belonging to the classes $M1$ and $M2$ respectively, represented by squares. The interaction between the sub-models is defined and implemented by means of a composition operator $OP$, represented by a rhombus. The figure also indicates the presence of data used to compose models (the pairs $e1_{M1}, e2_{M1}$ and $e1_{M2}, e2_{M2}$) that are exported or imported by means of interface elements. Some operators could not imply data flow, otherwise, the direction of the data flow is represented by oriented edges.
With reference to the example described in Section I-A, the model in Fig. 1 could be used to represent CN or SN nodes, where M1 could be the performance sub-model, M2 could be the availability sub-model and the operator OP describes the connection between them and the exchange of data that should be performed during the execution of the solution process.

This graph-based representation only shows the types of the models involved, the operators that must be applied, the existence of data flow and the topology of the composition. The bridge model also has a textual description (an XML document) containing the same information, and others that are not shown in the graphical representation.

The specific implementation of M1 and M2 is encapsulated in the class definition and the same holds for the composition operators.

Notice that the description of the bridge model and of its sub-models, the kind of analysis to perform on them, the required results and how to process them, the tools and their locations, must all be specified and managed. Moreover the problems arising from the usage of different input/output formats by multiple applications and tools must be addressed. The entire framework supporting all these activities and the execution of the workflow is briefly described below.

B. Orchestration Framework

The analysis of complex OMM models can be performed by means of the OsMoSys Multi-solution Framework (OMF) [2], [21], [22]. OMF was created to provide the support needed for a loosely coupled cooperation between heterogeneous analysis techniques and tools and automate the tasks that must be performed to solve complex multi-formalism models. Multi-solution in OsMoSys means solving a composed model according to a well defined solution process which involves the ordered execution of several solution or analysis tools (solversons). In other words, OMF achieves multi-solution by orchestration of solvers. A similar concept is implemented by Web Services orchestration [31].

Formalisms and models are described in OMF by a family of XML based languages [32]. A Query must be defined to solve a model. The Query contains a specification of the performance indices or measures that must be evaluated. A solution process is generated from the textual description of the bridge model and from the request specified by the Query and is then enacted by the OsMoSys workflow engine (WFE). The results obtained must be collected and processed to produce the answer to the Query defined on the model.

Notice that meta-modeling is not explicitly exploited at the OMF level. It is used to develop the multi-formalism models and build the bridge models, whereas the central role in model solution is played by the generation and the management of the workflow.

Fig. 2 shows the OMF architecture and the graphical user interfaces (GUIs) which are used to interact with OMF. The GUIs (represented by the three rectangles in the pale-gray box on top of Fig. 2) do not strictly belong to the OMF architecture (shown in the large dark-gray box).

The OMF architecture mainly consists of the following layers: 1) SPDL Compiler; 2) OsMoSys Core; 3) Solvers/Applications; 4) Repositories/DB. The SPDL Compiler automates the generation of the workflows, by using the model definition and the information about the current OMF configuration stored in the Repositories. The second layer contains the OMF components, which handle queries and results (Results Manager), instantiate models (Instancer) and enact the solution process (Workflow Engine). The third layer contains solvers and tools used to solve sub-models. Software modules (Adapters) wrap the existing tools in order to ensure a common interface with the solution process. An Adapter is used to manage different input/output formats, invoke a solver and handle results. If allowed by the solver, the Adapter can also interact with it during its execution. For example, this feature can be used in order to manage the state-space evolution of the model during its solution. The relationships between all the entities involved in the analysis of a model are described and handled by the Repository/DB layer.

The reader may refer to [20] and [22] for a more detailed description of the OsMoSys methodology and framework.

III. SPDL DEFINITION

In this Section the Solution Process Definition Language (SPDL) is defined and discussed. SPDL is a workflow language; therefore, in the following some basic workflow concepts are summarized and re-interpreted in terms of SPDL and solution workflows.

A. Workflow and SPDL Basic Concepts

According to the definition provided by the Workflow Management Coalition (http://www.wfmc.org/) a workflow is the automation of a business process, i.e. of “a set of linked activities or procedures which collectively realize a business
objective or a policy goal”. A workflow process definition is a network of activities and their relationships. Each activity represents one logical step within a process, i.e., the smallest unit of work to be performed. The completion of an activity and the starting of another activity is a Transition point in the workflow execution. A Transition may be unconditional, but the sequence of the activity execution may also be decided at runtime according to the value assumed by one or more logical expressions, in this case the sequence of operations depends on Transition Conditions that are evaluated after an activity has started or ended. The activation of an execution thread may be affected by the evaluation of the associated Transition Conditions.

Some points are defined within the workflow which allow the flow of the activities to be controlled: AND-split is a point in which a single thread of control splits into two or more threads which are executed in parallel. AND-join is a point in the workflow in which two or more parallel activities converge into a single thread of execution. XOR-split is a decision point where only one of alternative branches is executed. XOR-join is a point in the workflow in which two or more parallel activities converge in a single thread of execution without synchronization. OR-split is a decision point between several alternative workflow branches. OR-join is a point in which several alternative branches re-converge into a single threads of activities. In the following XOR, AND and OR are called split or join Conditions. These workflow elements are illustrated in Fig. 3.

![Workflow elements](image)

Therefore a workflow process definition itself consists of a network of Activity nodes, and it may be represented by a directed graph whose edges represent Transitions, and are a pair (FromActivity, ToActivity).

SPDL is the OMF workflow language used to define orchestrated solution processes for multi-formalism models. It consists of a network of activities that may refer to solvers or tools in order to enact the solution process.

The SPDL meta-model is depicted in Fig.4: its main components are: Participants, Applications, Activities, Transitions and Inner Variables.

Participants are the physical elaborating nodes where applications are executed. Activities comprise logical, self-contained units of work within the project. Four particular activities defined within SPDL are Route, Loop, Sub-process or Wait activities. Route activities allow for definition of complex control flow graphs, such as conditional branches etc. Loop activities allow for iterative execution of subprocesses. An activity may perform a set of tasks, invoking one or more application services. In particular, Applications are executed in order to perform models solution steps. Inner tasks perform actions needed during the solution process enactment that do not require external solvers or application execution. Expression evaluation is an example of such tasks. Attributes may be defined to specify activity control information and data used as input or output parameters (such as input or output files).

Inner variables are all the variables that may not be considered as results retrieved from sub-models solutions. For example, they can be used to define loop index variables or Transition Conditions. Inner variables can also be assigned at run-time during the solution processes enactment, like programming language variables.

Wait activities are used to manage external events and messages, or to allow for timed execution of other activities. Events in SPDL are persistent: an event “occurs” until it is “consumed” by a Wait activity. They can be generated by other activities before their termination, or they can be generated outside the process environment by users or by applications.

In order to be executed, a solution process has to be instantiated. This means that activities have to be bound to proper applications (if any) that have to be executed respecting the control flow defined by the solution process graph. An activity that is associated to one or more running applications or inner tasks is running and the pair (activity, running application/inner task) is an instance of the activity in the solution process definition. The enactment of a solution process requires activity instantiations in order to fulfill the solution process goals (i.e., to solve the model). Bindings between participants and applications can be performed both during solution processes definition and during processes enactment. In SPDL multiple activity instances can be concurrently executed.

![Solution process definition Meta-Model](image)

### B. SPDL Semantics

As previously stated, one of the most critical issues in the definition of the existing workflow languages is the lack of formal semantics. Therefore, after having described the main
Let $A$ be the set of the process Activities and $T$ the set of the process Transitions. In order to model the network of the Activities within the solution process, we introduce the functions $\pi_J$ and $\pi_S$, mapping an Activity to its join and split conditions respectively:

$$\pi_J : A \rightarrow \{\text{AND}, \text{OR}, \text{XOR}\}$$

$$\pi_S : A \rightarrow \{\text{AND}, \text{OR}, \text{XOR}\}$$

We denote $from(t)$ (to($t$)) the Activity having $t$ as outgoing (incoming) Transition and incoming($a$) = $[t_1, \cdots, t_n]$ (outgoing($a$) = $[t_1, \cdots, t_m]$) the set of the incoming (outgoing) Transition to the Activity $a$. Let $\sigma_A, \sigma_T$ be the functions mapping respectively an Activity and a Transition to their own states:

$$\sigma_A : A \rightarrow \{\text{running, terminated}\}$$

$$\sigma_T : T \rightarrow \{\text{fired, notFired}\}$$

The function $\pi_C$ maps an Activity to the number of instances (concurrently running) that will be created:

$$\pi_C : A \rightarrow \mathbb{N}$$

Let $t_{\text{begin}} = \{t_{\text{begin}_1}, \cdots, t_{\text{begin}_n}\} \subset T$ be the transitions set such that $t_{\text{begin}_i} \in \text{outgoing}\{\text{begin}\}$.

During the solution process execution, transitions can be activated or not. Once a transition is activated, we say that it is fired. At the process beginning, all activities are considered terminated, and all transitions are not fired. An activity can be then activated depending on the states of its incoming transitions and on its join condition.

Given the set of boolean predicates $P$ that are defined in the solution process, the partial function:

$$\lambda : T \rightarrow P$$

associates each Transition to the predicate that must be evaluated TRUE in order to make the transition to fire. Hence, $\lambda(T)$ is the Transition Condition set.

We can now introduce the function $\sigma_C$ mapping a transition to its truth value:

$$\sigma_C : T \rightarrow \{\text{TRUE, FALSE}\}$$

defined $\forall t \in T$, $\sigma_C(t) = \varepsilon(\lambda(t))$, where $\varepsilon : P \rightarrow \{\text{TRUE, FALSE}\}$.

As described before, an activity is running if and only if one or more of its instances are being executed. Let $\sigma_E$ be the function mapping an activity into the number of its executing instances, $\sigma_E : A \rightarrow \mathbb{N}_0$. $\forall a \in A$ the following relations hold:

$$\sigma_A(a) = \text{running} \Leftrightarrow \sigma_E(a) > 0$$

$$\sigma_A(a) = \text{terminated} \Leftrightarrow \sigma_E(a) = 0$$

Let $S = \sigma_A(A) \cup \sigma_T(T) \cup \sigma_C(T)$ be the States Set, consisting of the current values assumed by activities, transitions and conditions at the step $s$. Abusing notation, in the following we indicate by $S' = S(s)$ the States Set at the next step $s'$ during the workflow execution. The states changed and their new values are specified as arguments of $S$, the others ones are intended to be unchanged.

For example, see the rule (start) in Fig.5. The rule is an axiom. The consequence says that the States Set evolves from $S$ to $S'$, where in $S'$ all the transition belonging to the transition set $t_{\text{begin}}$ have been changed their value to fired (hence starting the workflow evolution), while the states of all the other activities, transitions and conditions remain unchanged.

The rules of the SPDL operational semantics are reported in Fig. 5. They are named by the labels used to identify them in the following description.

(start): The rule (start) states that all transitions outgoing from the begin activity can be fired immediately.

fire ($f_\lambda, f_v, f_w$): the three rules ($f_\lambda, f_v$) and ($f_w$), define the transition firing behavior depending on the split condition of its from($t$) activity.

termination ($t_{\text{Inst}}, t_{\text{Act}}$): An activity ends when its associated application or Inner Task terminates its execution. If one or more instances of the activity are still executing the activity is running.

Let $k \in \mathbb{N} : k \geq 2$ be the instance termination is modeled by the rule ($t_{\text{Inst}}$) and the activity termination by the rule ($t_{\text{Act}}$).

activation ($a_\lambda, a_v, a_w$): These rules model the activation behavior of the activity and wait activity. The three rules ($a_\lambda$), ($a_v$) and ($a_w$) state that an activity $a$ can become running depending on its join condition and on conditions of its incoming transitions:

- (AND) all its incoming transitions can fire;
- (OR) at least one of its incoming transitions can fire;
- (XOR) one and only one of its incoming transitions can fire.

At activation of an activity $a$, the number of instances associated to $a$ is set to the value $\pi_C(a)$.

Notice that, in the case of OR condition, if more than one transition can fire, all the incoming($a$) transitions become not fired as consequence of the activation of the outgoing activity $a$. An activity can, in fact, be activated more than once.

Unlike the activation of generic activity, the activation of Wait, Route and Loop activities are described as follows. In order to model the activation of the Wait Activity, we introduce the set $E$ of all the events that can occur during the enactment of the workflow process. Let $W \subseteq A$ be the set of wait activities. We define a function $\eta$ that associates every wait activity to the subset $E$ of events that must occur (at least one) in order to allow for activity activation: $\eta : W \rightarrow \wp(E)$.

The function $\sigma_O$ on the set $W$, maps a wait activity to TRUE if at least one event in $E$ has occurred:

$$\sigma_O : W \rightarrow \{\text{TRUE, FALSE}\}$$
Fig. 5. SPDL Operational Semantics

∀w ∈ WσO(w) = \bigvee_{e∈\varphi(w)} \varphi(e)

where

\varphi : E → \{TRUE, FALSE\}

The activation rules for the wait activity w ∈ W are reported in \((a_{w,\lambda})\), \((a_{w,\psi})\) and \((a_{w,\upsilon})\).

The route activity does not have any task associated with it and it terminates instantaneously.

Loop activity is quite different from the other ones. It contains a sub-process definition whose activities are iteratively instantiated and terminated with the same rules previously described. The sub-process state is not reset until the loop is finished. This activity also contains the definition of a function that is executed at the end of the cycle in order to assure loop condition evolution.

Loops are instanced when their JOIN conditions evaluates true. The activity terminates when its sub-process terminates (i.e. it reaches the end transition) and when the loop transition evaluates false. Otherwise, the sub-process is executed again.

C. SPDL Patterns Analysis

Different control flow constructs are defined on top of the concepts defined above: sequential routing, parallel routing, conditional routing; and complex structures maybe built by composing them. They have been widely studied and codified in forms of (workflow) patterns [33].

The main SPDL control flow patterns are depicted in Fig.6. They are listed in Table I where a comparison of SPDL with other workflow languages is shown.

The Sequence pattern can be implemented by linking two activities, as in Fig.6(a), regardless of the Join and Split conditions of A and B activities. The Parallel Split (Fig.6(b)) can be implemented by an activity whose split point leads to other activities by means of transitions whose conditions are evaluated true. The Synchronization pattern (Fig.6(c)) can be defined by a join point with AND condition where conditions of all incoming transitions evaluate true. Exclusive Choice can be implemented as in Fig.6(d) by defining an XOR split condition on the split point and defining outgoing transition conditions such that only one of them evaluates true. Simple Merge can be defined by joining two concurrent paths into an activity with an XOR join condition. The C activity is enacted once when the first activity between A and B terminates. The pattern is resumed in Fig.6(e). Fig.6(f) depicts the Multiple Choice pattern. It allows for the firing of multiple outgoing transitions (if their conditions evaluate true) by means of an OR condition at split point. The Structured Synchronizing Merge (Fig.6(g)) is defined as a composition
of Multiple Choice and Simple Merge patterns. The Multiple Merge pattern is similar to the Simple merge except for the D activity which is activated, in Fig.6(h) twice: once for each incoming path. This is implemented by defining an OR join condition on D. Discriminator is similar to the previous pattern except for the C transition that has to be fired only once as soon as the first of A and B activities terminates. This is achieved by defining an XOR join condition on C. The Arbitrary Cycles and the Structured Loops are implemented by using loop activities and loop transitions, which are fired depending on loop conditions. An example of this pattern is shown in Fig.6(i). The Explicit Termination pattern can be implemented by defining end activities in processes and sub-processes, while the Implicit Termination is implemented by the WFE when no more activities can be enacted during the execution of a workflow process. The Deferred Choice pattern is implemented as shown in Fig.6(j). There, the A activity has an XOR split condition. Since all transitions outgoing from this activity have conditions evaluating true, the choice of which activity to activate is deferred to the user by the WFE. The Interleaved Parallel Routing and the Interleaved Routing are implemented by disabling concurrent WFE scheduling. Persistent Triggers and Generalized AND-Join are trivially implemented by wait activities and AND Join conditions. The Local Synchronizing Merge is also implemented by using basic SPDL operators. Critical Sections can be implemented as shown in Fig.6(k), where the route activity is used in order to activate B or C activities assuring that the D−F and E−G paths are executed in a critical section. The route activity in the middle of the figure waits for the fire of a transition from start or H. The transition from start fires only at the beginning of the process enactment, assuring that the critical section is executed at least once. Further activations of the route activity follow the end of the H activity. Multiple Instances patterns that are supported by SPDL are shown in Fig.6(l) (without Synchronization), (m) (with a-priori Design-time knowledge) and (n) (with a-priori Run-time knowledge): the conc activity variable allows for multiple instances activation. Thread Merges and Splits are natively supported by the WFE.

Thanks to the formal definition of SPDL constructs, it is possible to prove that the reported pattern implementations reproduce the correct behavior.

In the following, a proof example is reported.

**Theorem 1 (Synchronization implementation soundness):**
The implementation of the synchronization pattern is sound in SPDL.

**Proof:** It must be proved that, applying the SPDL operational semantics rules if the C activity can terminate when A and B are running, the C activity can terminate. Two cases are possible: (a) A activity terminates after the B activity or (b) it terminates before. In both cases the C activity must become running (and then terminated) after the termination of the last between A or B.
solving can exchange during the enactment of the solution process and the Parameters that solvers needs to perform proper model analyses.

The differences between Messages and Wait Events lie in the way they are generated and managed: Messages are managed by adapters during their execution; Wait Events are managed by WFE and are related to Solution Process activities. Different classes of messages are used in order to: (i) exchange solvers state space variables and tool execution states (e.g. solver in execution, paused etc.); (ii) exchange information about the way the process variables can be retrieved (e.g. shared memories, files, sockets etc.); (iii) communicate commands for adapters executions (e.g. start, stop, pause etc.); (iv) support events during solvers execution.

It could be very complex (or impossible for some languages) to describe interaction patterns in workflow languages where these kinds of messages are managed. In SPDL it is possible to interact with proper adapters (by means of message passing), reading and writing solver variables and eventually state space variables during interactive simulation of models.

Activities in SPDL that execute adapters or other tools, may define a list of parameters to request correct solver executions. SPDL therefore has both a declaration and definition section where formal and actual parameters for solvers are listed.

Several solvers include interactive simulation as a model analysis mechanism, for example a “step by step” or a “run until condition or event” simulation. In such cases, it is possible to have access to the model state space at each simulation step. For this reason SPDL also allows the definition of (simple) “state-transition” state spaces. State elements can be exchanged among solvers in order to enact complex solution and simulation processes of multi-formalism models.

IV. GENERATION OF SOLUTION WORKFLOWS

In this section some hints are given about the automatic generation of SPDL specification and the role played by the composition operator in the automatic generation process. As explained in Section II, operators are used to build multi-formalism composed models. The composed models are in turn usually defined by acyclic oriented graphs.

If queries are defined on the model, it is possible to produce a stub for a solution process able to produce results for the defined queries. If all the operators in the composed model are data flow operators, the solution process can be expressed in terms of micro-operations, as described below. This work focuses only on the generation of solution processes in the presence of data-flow operators.

The syntax by which operators are expressed is the following:

\[(\text{OpName})(\text{LeftOperand},\\text{(c, : varName = RightOperand)}+,[\text{expr}])]+\]

where: OpName is the name of the operator; LeftOperand is one of the left operands of the operator; RightOperand is one of the right operands of the operator; varname is a name assigned to one RightOperand (this will be used in the expression); expr is an expression which processes the previous operands in order to assign

<table>
<thead>
<tr>
<th>Pattern</th>
<th>BPEL</th>
<th>XPDL</th>
<th>SPDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Parallel Split</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Synchronization</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Simple Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multi-Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Structured Synchronizing Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multi-Merge</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Structured Discriminator</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>Arbitrary Cycles</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Implicit Termination</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Explicit Termination</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multiple Instances without Synchronization</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>M. I. with a Priori Design-Time Know.</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>M. I. with a Priori Run-Time Knowledge</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>M. I. without a Priori Run-Time Know.</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Deferred Choice</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Interleaved Parallel Routing</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td>Milestone</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Structured Loop</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Recursion</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Transient Trigger</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Persistent Trigger</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Complete Multiple Instance Activity</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Structured Partial Join</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>Generalized AND-Join</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Static Partial Join for Multiple Instances</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic Partial Join for Multiple Instances</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Local Synchronizing Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>General Synchronizing Merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Critical Section</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Interleaved Routing</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td>Thread Merge</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Thread Split</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**TABLE I**

**PATTERNS EVALUATION**
a computed value to the \textit{LeftOperand}; \textit{cp}s are conditions used to determine which \textit{RightOperands} have to be used.

In the following $M_d$ is the destination model for data routing, while $M_s$ is the source model; $el_{ix}$ and $p_{el_{ix}}$ represent respectively an element and a property of an element in a model $X$. Let $r_{el_{ix}}$ be a result that can be retrieved after the solution on the $X$ model. The dotted notation proper of object-oriented languages will be used in the form: $M.el_{ix}.p$ denoting the property $p$ of the element $el_{ix}$ in the model $M$. In particular $M.el_{ix}.r$ is a result which can be obtained by analyzing the model $M$.

An operand is defined in terms of the model interface elements it addresses and of the properties involved in data manipulation operations. Operands, expressed in the form $M.el_{i}.p$ or $M.el_{i}.r$, must be interface elements: $M.el_{i} \in \mathcal{I}_M$.

The following is an example of operator definition:

$$(CER)M_{el_{i}.p}(M_{el_{i}.r} > = z) : (a = M_{el_{i}.r}; b = M_{el_{i}.r}, a + b)$$

$$(M_{el_{i}.r} < z) : (a = M_{el_{i}.r}; b = M_{el_{i}.r}, a - b)$$

where $M_{el_{i}.el_{j}.p}$ is used as the destination operand of the $CER$ operator. In the first case the sum of $M_{el_{i}.el_{j}.p}$ and $M_{el_{i}.el_{j}.r}$ is assigned to the value of the destination operand if $M_{el_{i}.el_{j}.r} > = z$, the difference otherwise.

Data-flow operators are resumed in Table II, where conditions and multiple operators are omitted for simplicity.

### Table II

<table>
<thead>
<tr>
<th>Full Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Properties (CPP) $M_{el_{i}.el_{j}.p_{M_d}}$</td>
<td>$(M_{el_{i}.el_{j}.p_{M_d}})$</td>
</tr>
<tr>
<td>Copy Results (CRP) $M_{el_{i}.el_{j}.r_{M_d}}$</td>
<td>$(M_{el_{i}.el_{j}.r_{M_d}})$</td>
</tr>
<tr>
<td>Copy Elaborated Properties (CEP) $M_{el_{i}.el_{j}.p_{M_d}}$</td>
<td>$(M_{el_{i}.el_{j}.p_{M_d}}, expr)$</td>
</tr>
<tr>
<td>Copy Elaborated Results (CER) $M_{el_{i}.el_{j}.r_{M_d}}$</td>
<td>$(M_{el_{i}.el_{j}.r_{M_d}}, expr)$</td>
</tr>
</tbody>
</table>

Copy Properties (CPP) and Copy Elaborated Properties (CEP) operators do not request a sub-model solution. They copy the value of the property of the source model into the property value of the destination model. In the case of the CEP operator, an expression that involves source model properties can be defined in order to assign elaborated values to the destination model property.

Copy Results (CRP) and Copy Elaborated Results (CER) instead request the solution of source models in order to use results to instantiate the destination model. The source properties have to be defined in $M_s$ formalism as properties that can be evaluated by solving the model (like timed transition throughputs in GSPN or station response time in QN). In the case of the CER operator, the solutions are pre-elaborated by processing $expr$ before being assigned to the specified destination property.

Classical programming languages type matching has to be satisfied for properties value assignments.

If conditions are defined, they usually involve properties or results evaluations. If results are needed, models also related to such results have to be solved, and results retrieved. Notice that results in $cp$s conditions are usually retrieved from the solution of the source models in the expression. Solution of models which are required for conditions evaluation have to be analyzed before the evaluation of all source models in the operation. This allows for a short-circuiting evaluation of the conditions.

In order to allow for automatic translations of composed models, information about formalisms elements and results are stored in the OMF repositories. This makes a clear distinction possible among elements properties and results simply by specifying only their (dotted) names.

### V. EXAMPLE: EVALUATING CAPABILITIES OF EMERGENCY RESPONSE

In this section, we exemplify our workflow approach by applying it to a homeland security scenario. The example is illustrative. The only purpose of the case study is to show how a multi-formalism model representing a real, complex system, may be effectively solved through the definition and the execution of SPDL workflows, and thus used to provide formal evidence of the properties of the modeled system. Therefore the scenario we describe in the following is developed in order to be plausible and, at the same time, to clarify how the workflows are generated and the experimental results are obtained, within the limits of space of this paper.

Therefore suppose that the goal of the model we want to develop is to evaluate local response capabilities in the event of an emergency or crisis, due for example, to a terrorist attack on a public site in which a monitoring and supervision system (MSS) may communicate data collected from sensors through a wireless network. The global system in this case consists of the site (MSS and network), public protection and other government agencies (fire brigade and rescue services, etc.). The occurrence of multiple events is considered, specifically a cyber attack to the network may attempt to delay the transmissions of data (used to state the kind of the attack: chemical, explosive, etc.) between the site and public protection units.

A common response requirement generated by this kind of scenario is the ability to treat victims at the scene, transport, treat and track patients. Here we emphasize this response thread by evaluating the capability of the system to direct, control, and coordinate a victim care response.

The Emergency Response Model (ERM for short) we use for the capabilities evaluation of the system is depicted in Fig.7.

It includes five sub-models connected through the CEP and the CER operators (see II). The formalism used to implement the single sub-model is specified in the figure in brackets under the class name. The core of the whole model is a public protection unit, which coordinates the operations, represented by means of a Timed Automata (Coordination PP-Unit). The site is modeled by two Generalized Stochastic Petri Nets representing the monitoring and supervision system (Site-MSS) and the communication network under attack (Tx-Att), respectively. Rescue services are included by modeling ambulances and hospitals by means of two Queuing Networks.
The parameters of the Site-MSS model are the type and the dimension of data collected from sensors. The model is used to evaluate the data throughput generated by the monitoring and supervision system.

This throughput is a parameter of the Tx-Att model which evaluates the real throughput of the data when a cyber attack is performed over the network. The other parameter of this model is the transmission rate of dirty packets on the network. Parameters of Ambulances and Hospitals include the number of available ambulances, the number of clear streets to travel, the number of hospitals and of emergency rooms (ERs) per hospital, the mean time needed to cover the distance to and from the hospitals (supposing that streets have been cleared) and the mean time it takes to give first aids in the ERs.

These models can be used to evaluate the average service time for people injured and to state if the planned resources (number of ambulances, emergency rooms and hospitals) can deal with the emergency. In addition, street utilization by ambulances can also be evaluated.

The Coordination model uses the (intermediate) results obtained by solving the other sub-models in order to evaluate if the deadlines are met in supporting the coordination and if the coordination procedures are correct.

The whole workflow solution process needed to solve the model as generated by the OMF is shown in Fig.8.

Activities and transitions are annotated by lists reporting the parameters used (white notes) and the results produced (grey notes).

In order to compact the process representation, the activities related to the solution of a model are clustered into sub-processes labeled by the name of the model to solve and they are depicted in Fig.8. The solution of a generic Model involves the steps depicted on the right of the figure, identified by the ~ symbol.

The solver tools involved in the solution process are Uppaal [35] for the Coordination model, SHARPE for Ambulances and Hospitals models, and GreatSPN for Tx-Att and Site-MSS models.

The Query introduced in Section II-B asks for the evaluation of the result throughput associated to some timed transition elements in the Tx-Att model. The answer to this Query is obtained by enacting the solution process and the related document is produced containing the values of the throughputs.

We briefly present and discuss some experimental results. The values of the main properties of the interface elements used in the reported experiments are listed below (we do not report all the parameters): (i) number of sensors in MSS (from 8 to 32 of several types), (ii) wireless network speed (11 Mbs), (iii) rate of dirty packets transmitted during the cyber attack (from 10 to 200 pkt/min.), (vi) number of hospitals (2, denoted H1 and H2), (v) number of ERs (10 in H1 and 8 in H2), (vi) number of point-to-point streets between the site and the hospitals dedicated to rescues (3), (vii) number of ambulances (from 1 to 10), (viii) point-to-point ambulance trip time from hospitals to the site or vice-versa (10 mins.), (ix) first medical aid time on site (4 mins.), (x) initial injured population (from 50 to 400 units), (x) discovery rate of newly injured (0.5 unit/min.).

Fig.9 plots the mean time that ambulances spend going from a hospital to the target site and back. Fig.10 shows the utilization percentage of the streets versus the number of ambulances (the increasing curves) and the mean load of the ambulances versus the street utilization (the decreasing curves).

Fig.11 shows the transmission time of the data generated by the MSS over the network versus the number of sensors and the intensity of the cyber attack (dirty packets per sec.). In the figure the number of sensors varies depending on the number of dirty packets sent by the attacker.

Finally, some properties checked on the Coordination model are reported in TableIII with the hypothesis that 12 sensors are configured in the Site-MSS. Ambulances may arrive at the site to provide first aid within 20 minutes. It is interesting to note the impact on this result of the cyber attack and of street utilization, when the number of ambulances increases (second row). Ambulances always arrive within 30 minutes. The results on the other rows do not depend on the cyber attack, which only affect the first phase of the rescues. After having
TABLE III
COORDINATION RESULTS

<table>
<thead>
<tr>
<th>Attack-Data Rate</th>
<th>Ambulances</th>
<th>Stations</th>
<th>Streets</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/100/200</td>
<td>NA</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>10/100/200</td>
<td>Yes/Yes/Yes</td>
<td>10</td>
<td>No</td>
</tr>
<tr>
<td>10/100/200</td>
<td>Yes/Yes/Yes</td>
<td>10</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Fig. 9. Ambulances trip mean time

Fig. 10. Utilization of ambulances and streets

Fig. 11. Transmission rate of MSS data

REFERENCES


Francesco Moscato is a Researcher and an Assistant Professor at Second University of Naples. His research activities are mainly centered on formal modeling and verification of reliable and critical systems. He is author of many articles published on international journals and books.

Flora Amato received her PhD in Computer Science and Engineering in 2009 from the University of Naples Federico II, where she teaches Computer Architectures. He led major research programs in conjunction with international universities, research agencies (CNR, ASI and EC) and technology leading industries, in Italy and abroad. He has a wide experience in the field of Complex Systems Modeling, Embedded Systems, General and Special Purpose Parallel Architectures and Performance Evaluation.

Antonino Mazzoco is Full professor of ”Computer Science” at the Computer and System Engineering Department, University of Naples “Federico II”. His research activities are mainly centered on: computer architecture, dedicated systems, reliable and secure systems, performance evaluation in high-performance systems. He authored over 270 papers on international journals, books and conferences.

Valeria Vittorini is Associate Professor at University of Naples ”Federico II”. She teaches Fundamentals of Computer Systems and Computer Programming. Her current research interests are in the area of Dependability and performance evaluation of computer systems. She is associate editors of the Int. Journal of Critical Computer Based Systems.