JoSEL: A Job Specification and Execution Language for Model Checking

Michael Westergaard and Lars Michael Kristensen

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {mw,kris}@cs.au.dk

Abstract. Model checking tools and techniques are being applied for verification of concurrent systems by users having different skills and background. This ranges from formal methods experts with detailed knowledge of the inner workings of the tools over students learning about model checking techniques to engineers that are mostly interested in applying the technology as a black-box. This paper proposes JoSEL, a visual language for specification of executable model checking jobs. JoSEL makes it possible to work at different levels of abstraction when interacting with model checking tools and thereby support the different kinds of users in a coherent manner. A verification job in JoSEL consists of tasks, ports, and connections describing the models to be verified, the behavioural properties to be checked, and the model checking techniques to be applied. A job can then be mapped onto an underlying model checking tool for execution. We introduce the syntax of JoSEL, informally define its semantics, and describe how it has been realised in the ASAP model checking platform.

1 Introduction

Model checking [1] is a very useful technique for validating the correctness of concurrent systems. Algorithms for performing model checking can be highly automated and a multitude of model checking tools have been implemented. In order to cope with the inherent state explosion problem, a large number of reduction techniques have been devised [9]. Reduction techniques typically exploit characteristics of systems to represent states more efficiently or to store only some of the states, but no reduction technique performs well for all systems. Also, many reduction techniques can be parametrised, and the chosen parameters can greatly affect the performance of the model checker. A useful model checker must therefore allow the user to select between a large number of reduction techniques and set parameters for each. Furthermore, new reduction techniques are constantly being developed, and a model checker must be able to incorporate new techniques easily. This means that it must be possible to augment the model checker, and that it must be possible for the user to access new techniques in a uniform way.

Model checkers can be used in at least three settings: by researchers (who develop reduction techniques), by students (who learn about model checking and reduction techniques), and by engineers (who analyse real-life systems). The three kinds of users have different backgrounds and approaches to using model checkers. Researchers may need extremely fine-grained control over all of the parameters, know in detail how the reduction techniques work, and often need to experiment with new reduction techniques and compare them to existing techniques. A student in the process of learning initially knows little about reduction techniques and wishes to experiment without pre-existing intuition about the techniques. An engineer will seldom learn the reduction techniques in detail so the model checker should basically be a push-button technology. Sometimes an engineer needs to fine-tune a few parameters. This means that users of a model checker work on different levels and need different abstractions. It is therefore desirable that a model checker makes it possible to hide details of the supported techniques, but also to fine-tune details when required.

The contribution of this paper is to propose the verification Job Specification and Execution Language (JoSEL) aimed at supporting users with different background when applying and experimenting with model checkers to solve concrete verification problems. A verification job in JoSEL consists of tasks, input/output ports, and connections. A job typically specifies model(s), properties to be verified, and model checking techniques to be applied. The tasks correspond to software components of an underlying model checker and the input ports of a task specify required parameters to the component. The output ports of a task specify the results produced when the component is executed. Output produced by one component can be used as input for other components by connecting the corresponding tasks using connections. Examples of components are storages, which can store reachable states, queues containing states that need to be processed, queries (e.g., in a temporal logic) that express behavioural properties, and hash-functions for storing states in hash-tables. JoSEL is independent of any concrete model checking tool, but for a verification job to be executed the tasks must be mapped onto components of a concrete model checking tool. This can be done in a manner which is fully transparent to the user, and in this paper we show how this has been implemented in the ASAP model checking platform [6].

JoSEL should be viewed as a flexible graphical alternative to hard-coding the supported model checking techniques into dialog boxes in a graphical user interface or relying on complex command-line arguments to manipulate the parameters of the model checker. To allow users to abstract away details that may not be required for everyone, JoSEL has macro tasks that basically represent a set of tasks and their interconnections. If a macro is given a meaningful name, it is possible to ignore the details of the compound task it represents. Still, it is easy to allow users to manipulate the details if needed. To make specifications created in JoSEL less prone to human errors, we assign types to the input and output ports of tasks. Finally, in JoSEL the user does not have to worry about the order in which tasks are to be executed as we can do a simple dependency analysis and execute components in a correct order.

It is possible to specify verification tasks using other techniques, e.g., a textual description like shell scripts, Makefiles or custom scripting languages, such as

SVL [4]. Textual languages have several disadvantages making them difficult to use for inexperienced users. Firstly, the user needs to remember keywords and names of the available components, and, secondly, textual languages are very prone to typing errors. Furthermore, and more importantly, textual descriptions have a built-in order, and it may be confusing if it does not correspond to the order of execution. Hence, at least at some level, users need to worry about the order in which the components are executed. Graphical alternatives exist as well. One example is a workflow specification language [10], but they are not really tailored to dealing with model checking, and we find that a language designed specifically for model checking is preferable. An example of a language for this purpose is jETI [7], whose main objective is to easily compose web-services. jETI does not, however, make it easy to specify several similar tasks, and is more tailored to a write once, run many times paradigm, whereas we seek a language that makes it possible for experienced users to make building blocks that can be used and modified by less experienced users that are not necessarily programmers. JoSEL addresses the above problems. As JoSEL is a graphical language it can show users all available components and only allow the user to select legal components, so the user does not have to remember the names of the components, thereby alleviating the problems with textual languages. JoSEL also takes care of the order of execution so that all input values required for each part of a task are guaranteed to be available. Furthermore, JoSEL is designed with abstraction mechanisms that allow experienced users to design components that can be used directly as black boxes or adapted by other users.

The rest of this paper is structured as follows: Section 2 introduces the constructs of JoSEL and defines its syntax. In Sect. 3, we informally define the semantics of JoSEL and indicate how we have realised JoSEL in the ASAP model checking platform. We sum up our conclusions in Sect. 5.

2 Syntax

The Job Specification and Execution Language (JoSEL) is inspired by workflow specifications [11] and data-flow graphs [8]. We do not use these languages directly as we aim at a language tailored for specification of verification jobs. Furthermore, we want our job specifications to be executable which means that we must be able to distinguish the inputs and outputs of processes, and we need to be able to instantiate and synchronise processes based upon the data content. The two latter requirements means that we cannot use data-flow graphs directly. In this section, we formally introduce the graphical syntax of JoSEL; in the next section we consider the execution of JoSEL specifications.

To introduce JoSEL we use an example of a verification job where we want to verify two safety properties of a given (formal) model. The properties we wish to check are that the model contains no dead-locks (states without enabled transitions), and that it is impossible to reach a state where a certain buffer overflows. We assume that dead-lock freeness is a standard property built into the underlying model checking tool, and that the buffer overflow property can be expressed as a state predicate in a propositional logic. Executing this job consists of systematically traversing all reachable states of the model while storing already encountered states in a storage. If a violation is found, i.e., if we encounter a dead-locked state or a state where the buffer has overflowed, an error-path leading to the violating state should be reported to the user. We introduce JoSEL by constructing the example job bottom-up from scratch, i.e., we deal with the most specific details first. In a real model checker, it is rarely required to specify all parts as some building blocks would be available beforehand. We build the job to illustrate how JoSEL supports accessing the detailed parameters of the model checker components when required.

2.1 Tasks

The basic unit of computation in JoSEL is a *task*, which corresponds to a process in data-flow graphs. The reason for the different terminology is in part due to workflow specifications, whose terminology we have adopted, and due to the fact that a task can be instantiated more than once when a job is executed, thereby giving rise to multiple *processes*. As we need to graphically fit information for inputs and outputs into each task, we do not use a circle to represent tasks, but rather a rounded rectangle. Figure 1 shows a task named Instantiate Hash Table Storage. The task in Fig. 1 is able to instantiate the storage required for the verification job.

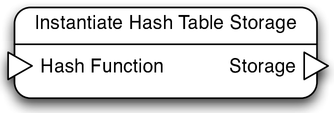


Fig. 1: A task with one input port and one output port.

Associated with tasks are two kinds of *ports*: *input ports* and *output ports*. Input ports specify the input data required for a task and are by convention located on the left-hand side of a task. Output ports specify the output data produced by a task and are located on the right-hand side of a task. Ports are represented by triangles; input ports point into the task, symbolising values going into the task, and output ports point out of the task. The name of a port is written inside the task next to the port itself. The task in Fig. 1 has one input port and one output port. The set of input ports of a task is denoted I and the set of output ports of a task is denoted O. In the example we have $I = \{\textsf{Hash Function}\}$ and $O = \{\textsf{Storage}\}$. This specifies that in order to generate a storage (the output port) that store states in a hash table, we need a hash function (the input port).

Associated with each port of a task is a *port type* specifying the kind of data that can be consumed/produced at the port. This is represented by a mapping

$\tau : P \rightarrow \Sigma$ from the set of all ports $P = I \cup O$ of the task to a set of types Σ. The actual types allowed are implementation specific, but can, e.g., be simple strings, integers and files, or more complex data-structures like hash-tables or representations of formal models and state space graphs. The type of each port is not reflected in the graphical representation, since in practice the type of a port can be inferred from the name of the port and task. In the example in Fig. 1, it is fairly clear that the input has a type that is a hash function and the type of the output port is storage. The main reason for omitting the types from the graphical representation is simplicity of the drawing, but tools implementing JoSEL are free to reveal the types if desired. Associated with each port is also a *port mode* which is either unit, iterator, or collection. The port mode describes how data is consumed (input ports) and produced (output ports), and is specified by a port mode mapping PM. Basically, unit ports only produce/consume one value, whereas iterator and collection ports can consume/produce more than one value. We explain port modes in more detail in Sect. 2.4 and Sect. 3 when discussing the execution of tasks. The following summarises the definition of tasks.

Definition 1. *A* ***task*** *is a tuple* $T = (I, O, PM, \Sigma, \tau)$ *where:*

- I *is a finite set of* ***input ports****,*
- O *is a finite set of* ***output ports*** *such that* $I \cap O = \emptyset$*,*
- $PM : P \rightarrow \{\mathsf{unit}, \mathsf{iterator}, \mathsf{collection}\}$ *assigns to each port a* ***port mode****, where* $P = I \cup O$*.*
- Σ *is a set of types and* $\tau : I \cup O \rightarrow \Sigma$ *assigns to each port a* ***port type****.* □

2.2 Jobs

When we have a task able to generate a hash function for use by the task in Fig. 1, we would like to be able to specify that the hash function produced by that task should be passed to the Instantiate Hash Table Storage task. A *job* consists of a set of tasks $\mathcal{T}$ connected by a set of *connections* C describing how the output produced by one task is used as input to other tasks. A connection is a pair (c_O, c_I) connecting an output port c_O of one task with an input port c_I of another task. We represent a connection by a line from the output port to the input port. In Fig. 2, the output port of Instantiate Hash Function is connected to the input port of Instantiate Hash Storage. In this case Instantiate Hash Function is able to generate a hash function. This is done from a specific model that must be provided. The generated hash function is passed to Instantiate Hash Storage, which can generate a storage, which is able to store states of the model given to Instantiate Hash Function. Input ports can be connected to output ports with the same type. It is, however, easy to allow sub-typing, by loosening this requirement such that the type of the output port is only required to be a sub-type of the type of the input port. This is also how we have implemented it in ASAP [6] (using the type hierarchy of Java as the sub-typing relation), but we have omitted it in the formal definition for simplicity. Furthermore, we require that the directed graph induced by the tasks and connections in a job is acyclic

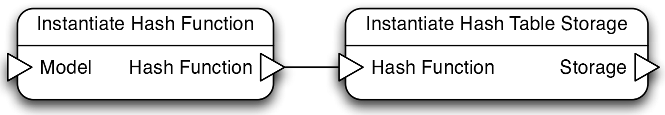


Fig. 2: A job consisting of two tasks.

to ensure termination. As we require that input ports are always to the left and output ports are to the right of the task, this naturally leads to a flow from the left to the right of the job. The following summarises the definition of a job:

Definition 2. *A* ***job*** *is a tuple $J = (\mathcal{T}, C)$ where:*

- *$\mathcal{T} = \{T_i = (I_i, O_i, PM_i, \Sigma_i, \tau_i)\}_{1 \le i \le n}$ is a finite set of* ***tasks*** *satisfying $i \neq j \Rightarrow P_i \cap P_j = \emptyset$, where $P_i = I_i \cup O_i$*
- *$C \subseteq \mathcal{O} \times \mathcal{I}$ is a set of* ***connections*** *(where $\mathcal{I} = \bigcup_{T_i \in \mathcal{T}} I_i$, $\mathcal{O} = \bigcup_{T_i \in \mathcal{T}} O_i$) satisfying that the directed graph induced by tasks and connections is acyclic,*
- *For all $c_O \in O_i$ and $c_I \in I_j$ such that $(c_O, c_I) \in C$ we have $\tau_i(c_O) = \tau_j(c_I)$.*

$\square$

It should be noted that the definition allows multiple connections to and from each port. This is, e.g., useful if we want to use the value produced at an output port in multiple locations or if we want to consume values from multiple sources or instantiate processes for values calculated by different processes. It is possible for tasks in a job to have *free ports*, i.e., ports that are not assigned via connections. The set of free input ports of a job J is denoted $\mathit{Free}_I(J)$. A job is *closed* if it has no free input ports, otherwise it is *open*. The job in Fig. 2 is open as the input port Model of Instantiate Hash Function is free.

2.3 Macro Tasks

In order to support different levels of abstraction, we introduce *macro tasks* (or macros for short). A macro task is a high-level representation of a job with free ports. A macro task can be thought of as a component of a job which is intended to be re-used in different settings and therefore does not have all parameters defined immediately. For example, we may want to reuse the job from Fig. 2 in several different job descriptions, as it provides an implementation of a way to generate a storage that is able to store states of a given model. Instead of copying the entire job, we just draw a special macro task. A macro task is graphically represented like an ordinary task except that we draw its outline using double lines. An example of a macro task can be seen in Fig. 3 (top). In order to specify the input and output ports of a macro, we introduce a special kind of port, which states that when a job is used as a macro, this port should be available to the user of the macro. Such ports are called *exported ports* (exported input/output ports), and are drawn using a double outline, as in Fig. 3 (bottom), where Model and Storage are exported. Outside the task, next to an exported port,

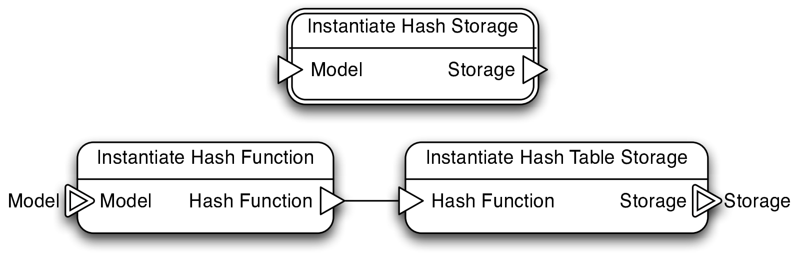


Fig. 3: A macro task (top) and the corresponding job (bottom).

we write the *exported name* of the port. The name is chosen by the user, but will often be the same as the name of the port. The macro task in Fig. 3 (top) can represent the job in Fig. 3 (bottom). The exported name becomes the name of the port in the macro. Macros are purely syntactical, and can be removed by repeatedly replacing macro tasks with the tasks of underlying jobs, moving connections to/from a port on a macro task to the corresponding exported port in the job, and removing all exported ports. We require that the set of exported input ports includes all the free input ports of the job and that the exported ports are contained in the ports of the underlying job. Port modes and types of an exported job are inherited from port modes and types of the underlying job. The following summarises the definition of a macro task.

Definition 3. *Let $J = (\mathcal{T}, C)$ be a job and $\mathcal{T} = \{T_i = (I_i, O_i, PM_i, \Sigma_i, \tau_i)\}_{1 \leq i \leq n}$. J can be represented by a* ***macro task*** *which is a task $M = (I, O, PM, \Sigma, \tau)$ where:*

- *I is a set of* ***exported input ports*** *such that $I \subseteq \mathcal{I}$ and $Free_I(J) \subseteq I$, where $\mathcal{I} = \bigcup_{T_i \in \mathcal{T}} I_i$.*
- *O is a set of* ***exported output ports*** *such that $O \subseteq \mathcal{O}$, where $\mathcal{O} = \bigcup_{T_i \in \mathcal{T}} O_i$.*
- *$PM(p) = PM_i(p)$ for all $p \in P \cap P_i$ and $1 \leq i \leq n$, where $P = I \cup O$ and $P_i = I_i \cup O_i$.*
- *$\Sigma = \bigcup_{1 \leq i \leq n} \Sigma_i$ and $\tau(p) = \tau_i(p)$ for all $p \in P \cap P_i$ and $1 \leq i \leq n$.* □

2.4 Hierarchical Jobs

In our running example we have until now constructed a means to store states of a given formal model (cf. Fig. 3). We now hierarchically construct the remainder of the verification job for checking the two safety properties. The next step is to construct a job for checking a safety property of a model. Such a job is shown in Fig. 4. The job has a Waiting Set Exploration task which traverses all states, stores all states we have already visited in a Storage, and stores all discovered but not yet processed in a Waiting Set. The task is parametrised to allow flexibility. For example, we can use a storage storing states in a balanced tree or on disk instead, or we can use a different waiting set to impose a different traversal order. The typing of ports makes sure that only storages and waiting sets that are actually usable can be connected. Our previous macro task from Fig. 3, Instantiate Hash Storage, takes care of creating a storage, and the task Instantiate Queue takes care of instantiating a specific waiting set, implemented as a queue, which imposes a breadth-first traversal of the states. As the model has to be used both for the Waiting Set Exploration and as input for instantiating the storage and waiting set, we use a Multiplex task, which just lets its input flow unaltered to the output port. Here we have used the ability to connect an output port to multiple input ports. The result of the Waiting Set Exploration is a Traversal, an abstract result, which can be used by the On-the-fly Safety Checker to check properties. We see that the job in Fig. 4 exports four ports, Model, Properties, Answer, and Error trace. The Instantiate Queue, Waiting Set Exploration, and On-the-fly Safety Checker are all components of the underlying model checker.

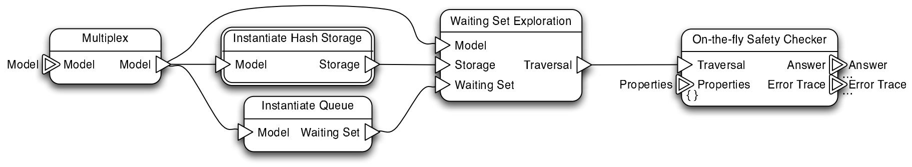


Fig. 4: Specification of a verification job for checking safety properties.

The exported port Properties in Fig. 4 has port mode *collection*. We indicate that a port has mode collection by annotating it with "{ }" (like a set). On-the-fly Safety Checker takes care of actually checking the properties given. It produces an Answer (a boolean value; did the property hold) and a path to violating states (if any). In order to do this efficiently, it needs the set of all safety properties when the task is started which is exactly what collection ports specify. As the On-the-fly Safety Checker task can take more than one property as input, it may also produce more than one output for both Answer and Error trace. We could assign both of these mode collection, but it is possible that we can provide an answer for one of the safety properties earlier than we can for other properties. The Answer and Error trace ports therefore have port mode *iterator*, which indicate that multiple outputs may be produced, and that they are produced one at a time as they become available. Iterator ports are annotated with "...". Letting Answer and Error trace be iterator ports allows us to show a user that one of the properties has been violated before the execution of the task has completed. We also allow iterator ports to be input ports (indicating that we can consume values one at a time) and collection ports to be output ports (indicating that all results are returned when the task has completed). Ports that are neither collection ports nor iterator ports are *unit* ports which means that they consume/produce exactly one value per task instantiation.

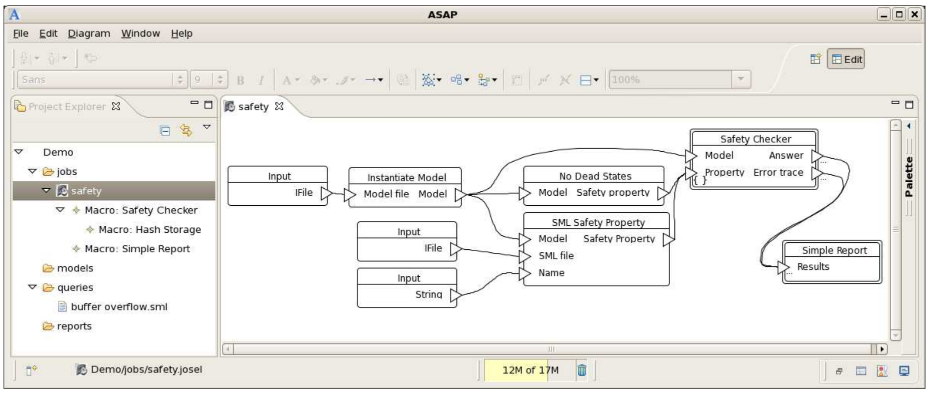


Fig. 5: Using the ASAP tool to model check using JoSEL.

To check the two safety properties for a given model we create the job in Fig. 5 which constitute the *root job* of our example verification job. Figure 5 is a snapshot from the implementation of JoSEL in ASAP. The Safety Checker macro task represents the job in Fig. 4. In addition, we have tasks to instantiate a safety property for checking dead-locks (No Dead States), and to instantiate a safety property from a user-supplied file (for checking buffer overflow; two Inputs, one for specifying the file and one to give a descriptive name of the property, and SML Safety Property). Both of these properties are input to the Safety Checker. Additionally, we have tasks to instantiate a model from a user-supplied file (Input and Instantiate Model). The model is also passed to the Safety Checker. Furthermore, we want to generate a report which shows whether the properties were violated and, if so, how we can reach a state where the property is violated. At our disposal we have a Simple Report macro which is able to generate such reports. The task is passed the values produced by the Safety Checker. The Simple Report task is able to update the report as answers arrive, so we can see partial results during the calculation. The Simple Report macro is an example of connecting more than one output port to a single input port. We need not worry about the actual implementation of the Simple Report macro as long as we are aware that it creates a report and displays the values passed to it. The Simple Report task is in fact implemented so that it is possible to extend its capbilities to show values of various types, but, as the actual operations of the task is out of scope of this paper, we can just assume that the task is able to perform reasonable post-processing and display any value passed to it.

Our example shows that JoSEL can express rather complex jobs at different levels of abstraction. The top-level view (Fig. 5) allows us to specify which properties to check of which model and what to do with the results. The next level (Fig. 4) allows us to determine the traversal strategy (using different waiting set implementations) and how to store states (by using another storage implementation). At this level we can also swap the safety checker for, e.g., an LTL model checker, which is able to check more complex properties. At the lowest level (Fig. 3) we can change which hash function is used to store states in a hash table. The three levels together forms a *hierarchical job* as defined below.

Definition 4. *A **hierarchical job** is a tuple $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ where:*

- *$\mathcal{J} = \{J_i = (\mathcal{T}_i, C_i)\}_{1 \le i \le n}$ is a finite set of jobs such that $\mathcal{T}_i \cap \mathcal{T}_j = \emptyset$ and $C_i \cap C_j = \emptyset$ for $i \neq j$.*
- *$J_r \in \mathcal{J}$ is a distinguished **root job**,*
- *$\mathcal{T} = \{T_k = \{I_k, O_k, PM_k, \Sigma_k, \tau_k\}\}_{1 \le k \le m}$ is a set of macro tasks such that for all $T_k \in \mathcal{T}$ there exists a $J_i \in \mathcal{J}$ such that $T_k \in \mathcal{T}_i$,*
- *$SJ : \mathcal{T} \to \mathcal{J}$ assigns jobs to macro tasks such that SJ is onto $\mathcal{J} \setminus \{J_r\}$, T_k is a macro task for $SJ(T_k)$ for all $T_k \in \mathcal{T}$ (cf. Def. 3), and the graph induced by $\mathcal{J}$ and SJ is a tree rooted in T_r.*
- *For $J_i, J_j \in \mathcal{J}$ with $i \neq j$, $(\mathcal{P}_i \setminus \mathcal{P}'_i) \cap (\mathcal{P}_j \setminus \mathcal{P}'_j) = \emptyset$, where $\mathcal{P}_i$ are the ports of tasks of J_i and $\mathcal{P}'_i$ are the ports of macro tasks of J_i.* □

The last requirement in the above definition ensures that only exported ports are shared between jobs. A hierarchical job can be flattened by replacing all macro tasks with the corresponding jobs and moving the connections to the correct ports as defined below.

Definition 5. *Let $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ be a hierarchical job with $\mathcal{J} = \{J_i = (\mathcal{T}_i, C_i)\}_{1 \le i \le n}$. The corresponding **flattened job** is $J = \{T, C\}$ where $T = \bigcup_{1 \le i \le n} \mathcal{T}_i \setminus \mathcal{T}$ and $C = \bigcup_{1 \le i \le n} C_i$.* □

A hierarchical job is said to be closed when the root job is closed. Whenever a hierarchical job is closed, so is the corresponding flattened job as stated in the following proposition.

Proposition 1. *If $H = (\mathcal{J}, J_r, \mathcal{T}, SJ)$ is a hierarchical job. Then the corresponding flattened job is closed if and only if H is closed.*

Proof. Any free input ports of a job $J_j = SJ(T)$ with $T \in \mathcal{T}_i$ is either connected in $\mathcal{T}_i$ or a free input port of $\mathcal{T}_j$ (Def. 3). Thus any free port of J_r is a free port of the flattened job as no connections are broken during flattening, and H is closed if and only if J_r is closed. □

3 Execution of JoSEL Specifications

In this section we outline the semantics of JoSEL and outline how this has been implemented in the ASAP model checking platform [6]. The semantics define the dynamics of the language. We can formally define the semantics, but for an intuitive understanding of JoSEL this is not necessary, and may in fact be counter-productive due to the level of detail required to fully specify JoSEL. Instead we intuitively describe how jobs are executed with particular focus on how instantiation works when ports with different modes interact.

The basic idea in the semantics of JoSEL is to instantiate each task such that whenever a task is instantiated, all preceding tasks (tasks with a connection to an input port of the task in question) have been instantiated and completed (at least all values for non-iterator input ports). For non-hierarchical jobs, this can be ensured by performing a topological sorting of the tasks where a task $T_1 = (I_1, O_1, PM_1, \Sigma_1, \tau_1)$ is sorted before $T_2 = (I_2, O_2, PM_2, \Sigma_2, \tau_2)$ if there is a connection (c_O, c_I) from T_1 to T_2 ($c_O \in O_1$ and $c_I \in I_2$). As this graph is acyclic (cf. Def. 2) this sorting is well-defined and yields a total order of all tasks (tasks that using the aforementioned ordering are equal or incomparable are simply taken in an arbitrary order). If we further assume that the job is closed, executing jobs according to this order will ensure that all values are available when we want to execute a task. If we are given a hierarchical job, we construct (at least conceptually) the corresponding flattened job according to Def. 5. Due to Prop. 1, we easily see that a hierarchical job can be executed when it is closed.

3.1 Port Modes and Instances of Tasks

If ports with different modes are connected, we may either instantiate a task more than once or synchronise a number of tasks depending on the port modes. The effects of all possible combinations of ports are summarised in Table 1. Basically, whenever an input port has mode unit, we start an instance for each value it is given. If we have multiple connections with different modes to an input port with mode unit, we simply start a thread for each value arriving from each connection, essentially implementing a fork semantics. Similarly, we always pass exactly one collection or iterator to input ports with mode collection and iterator, respectively. The result of having multiple connections to such a port is thus to create exactly one collection containing all data or one iterator iterating over all values. Iterator output ports differ from collection output ports in that values can be passed before the task producing them has terminated. This is, e.g., useful for data collection. Table 1 shows what happens when a single output port is connected to a single input port.

Based on the intuition of the semantics, we obtain the following proposition which states that if the execution of all tasks terminate for all inputs and produce only a finite amount of data, execution of the job always terminates.

Proposition 2. *Let H be a closed hierarchical job. If all tasks of jobs in H terminate for any input and all iterators produce only a finite number of data-elements, the execution of H eventually terminates.*

Proof. The proof is in the structure of the flattened job corresponding to H. If we assume that the predecessor of a task T produce only a finite number of data elements and that predecessor tasks can only be instantiated a finite number of times, T can only be instantiated a finite number of times. $\square$

3.2 Implementation in ASAP

The outlined semantics assume that we wait for all instances of a task to run to completion before starting subsequent tasks. As most modern computers have

Table 1: The effect of connecting different kinds of ports.

Output	Input		
	▷ Unit	▷ Iterator ...	▷ Collection {}
Unit ▷	Directly pass value	Create iterator over one value	Create a collection with one value
Iterator ▷ ...	Fork as values arrive	Pass all values to a single task instance as they arrive	Synchronise; wait until all values have arrived and pass in a collection
Collection ▷ {}	Fork for each value in collection	Create iterator over collection and pass to a single task instance	Pass all values to a single task instance

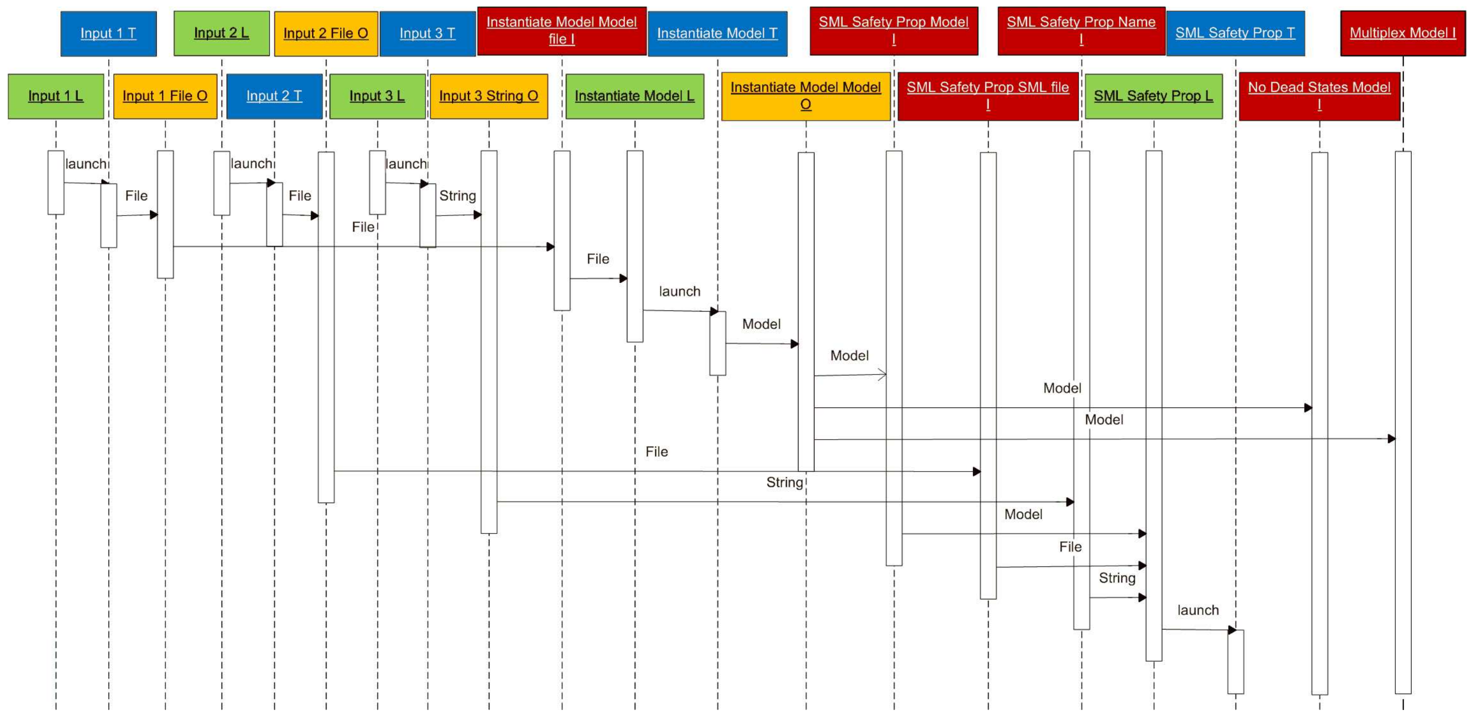
CPUs with two, four, or more cores, this is not optimal. Instead, we have made a more efficient implementation in ASAP, which basically allows tasks to start as soon as all the input values they require are available. From Prop. 2, we see that this strategy eventually will lead to executing all tasks.

The implementation of JoSEL in ASAP is done by implementing the data-structures corresponding to the tuples in the previous section using the Eclipse Modeling Framework (EMF) [3], implementing a simple object oriented representation of tasks, connections, and jobs. On top of that, we have developed a graphical editor using the Eclipse Graphical Modeling Framework [2], which semi-automatically generates a graphical editor that automatically maintains the object oriented representation of JoSEL, so we only need to consider an abstract representation of JoSEL when dealing with the execution of jobs. As we only deal with the abstract representation it is in principle possible to develop other concrete syntaxes for JoSEL if one, e.g., prefers a textual syntax.

Our implementation of execution of JoSEL specifications aims to increase concurrency and observes that starting many threads that are just waiting to be executed is relatively inexpensive. The idea is to create a thread for each task (a *task thread*) which is intended to contain the actual implementation of the task, a thread for each output port (an *output thread*) to distribute the output to multiple recipients, a thread for each input port (an *input thread*) to collect and synchronise input from multiple sources and put the data to the correct format, and finally a thread to spawn the appropriate task thread and pass parameters (a *launcher thread*). We may start multiple instances of a task thread, one for each execution of the task, but only one instance of each output, input, and launcher thread. Additionally, we have a communication channel for each output port, allowing task threads to transmit values to the correct output threads, a channel for each connection, allowing output threads to transmit data to the correct input threads, and one for each input port, allowing input threads to pass data to the correct launcher thread. Finally, We need to keep track of how many instances of each task thread we have started and when we will start no more task threads for a specific task thread. This information allows subsequent output threads to realise when they have received all output values from task threads.

In the figure on page 14, we see some of the threads instantiated in ASAP to execute the JoSEL specification in Figs. 3, 4, and 5. Threads are named after their task and, in the case of input and output threads, after their port as well. Threads are identified by an L, a T, an I, or an O for launcher, task, input, and output threads, respectively. We have numbered the three Input tasks in Fig. 5 from the top to the bottom in order to better be able to distinguish them. Active threads are marked by a white band. We notice that in the beginning all threads except for task threads are active (but waiting). Initially, the launcher thread for the Input 1 task realises that it has all input values required to start an instance of the task, so it launches a task thread for the task. This thread runs to completion and produces a single output value, which is transmitted to the output thread for the File output port of the Input 1 task. Independently, the launcher, task, and output threads for the Input 2 and Input 3 task do the same thing. The single connection from the File output port of the Input 1 task to the Model file input port of the Instantiate Model task, makes the output thread transmit the value to the input thread of the Model file input thread, which is then awoken. The value needs no translation, so it is just transmitted directly to the launcher thread of Instantiate Model, which realises that it has received enough input values to launch an instance of the Instantiate Model task thread. This runs and produces a Model output value, which is transmitted to the output thread. The corresponding output port has three connections, so the value is transmitted to the three corresponding input threads. At some point, the File and String values have been transmitted to the SML file and Name input threads of the SML Safety Property task. The three input threads transmit the values to the launcher thread of the SML Safety Property task, which realises, it has enough information to launch a SML Safety Property task thread. The computation continues in this manner.

As indicated from the example the threads collaborate to perform the entire computation specified by the specification. The different functions of the different kinds of threads have been summarised in Table 2. Task threads perform the actual execution of a single instance of a task. A task thread receives a correctly formatted value for each input port, executes the task with the given values and transmits values produced for each output port to the corresponding output process. Output threads take care of transmitting values to the correct input threads, and as such implement the connections. When there is only a single connection from an output port, this just consists of passing on the value. In the case where there are more outgoing connections (such as the Model output port of the Instantiate Model task in Fig. 5), the value must be copied, in particular if it is a port with collection or iterator mode. Input threads are responsible for receiving values from multiple output threads (as in the case of the Properties input port of the On-the-fly Safety Checker in Fig. 4, which is exported and connected to multiple output ports in Fig. 5). Input threads are also responsible for


Input 1 T
Input 2 L
Input 2 File O
Input 3 T
Instantiate Model Model file I
Instantiate Model T
SML Safety Prop Model I
SML Safety Prop Name I
SML Safety Prop T
Multiplex Model I
Input 1 L
Input 1 File O
Input 2 T
Input 3 L
Input 3 String O
Instantiate Model L
Instantiate Model Model O
SML Safety Prop SML file I
SML Safety Prop L
No Dead States Model I
launch
File
launch
File
launch
String
File
File
launch
Model
Model
Model
Model
File
String
Model
File
String
launch

converting the received values to the correct format. In the example of the Properties input port of On-the-fly Safety Checker, the input port has mode collection, but it receives values from two output ports with mode unit. The input thread must therefore construct a collection containing the values received. A similar conversion must take place when the input port has mode iterator. If the input port has mode unit, values are simply transmitted one at a time as they are received, optionally extracting values from collections or iterators. Finally, input value(s) on the correct form are transmitted to the launcher thread. Launcher threads receive correctly formatted input values and as soon as a value is received for each input port, a new task thread is started with the correct parameters. As more than one value can be received on unit input ports (collection and iterator ports always receive exactly one collection or iterator), it is possible that more than one task instance needs to be started. The launcher thread takes care of starting new task threads as soon as possible by constructing new elements of the Cartesian product of the input values as soon as possible.

Table 2: The function to be performed by each kind of thread.

Thread	Function
Task	Run a single instance of the task with the provided parameters and transmit the produced results to the correct output threads.
Output	Receive values and transmit them to all input threads according to the connections.
Input	Collect values from multiple output processes and put them into the correct form according to the input port mode.
Launcher	Receive values from input threads and launch new task threads whenever enough values have been received to provide a full set of inputs.

4 Example

Let us see some examples of use of JoSEL in ASAP from the point of view of different users. Assume a person from the industry wishes to verify a concrete model of a network protocol. The task is to verify that the protocol contains no dead-locks and that the buffers of the participants do not overflow. As the model is of a real-life system, it is suspected that the state space may be large, and as the protocol works in steps, it is decided, guided by ASAP, to use the sweep-line state space reduction method, which exploits progress to only store parts of the state space in memory at any time. A standard JoSEL template specification for checking for dead-locks using the sweep-line method is created using a simple wizard, and a specification like the one in Fig. 6 is obtained (for legibility, we have removed all the input tasks; all unassigned input ports are connected to the output of a corresponding input task). Compared to the example from Fig. 5 the

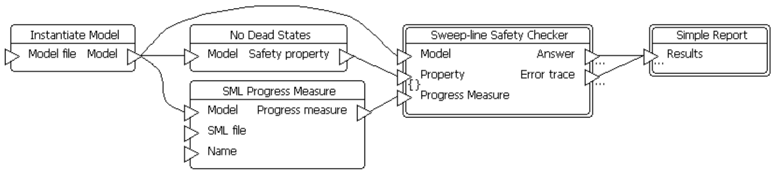


Fig. 6: Initial dead-lock freeness checker using sweep-line method.

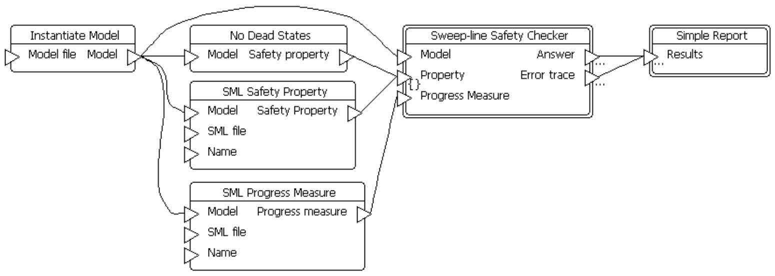


Fig. 7: Dead-lock and buffer overflow checker using sweep-line method.

only difference we see is that there is no SML Safety Property task and instead we have a SML Progress Measure, which is used by the sweep-line method to decide when states no longer need to be kept in memory.

Our industry person slightly modifies the task and adds tasks to specify the no buffer overflow property. The task now looks like the one in Fig. 7, which identical to the one in Fig. 5 except for the added progress measure. Another way to arrive at the job in Fig. 7 would be start from Fig. 5, observe that the verification is not able to go through as too much memory is consumed, and delete the Safety Checker macro and replace it with a Sweep-line Safety Checker built into the tool.

Assume now a student wants to experiment with the sweep-line method. The student starts with the same job from Fig. 6 and digs into the details of the Sweep-line Safety Checker and sees the implementation shown in Fig. 8. Compared to the implementation of the Safety Checker in Fig. 4, we have removed the Queue and use a Sweep-line Exploration instead of a Waiting Set Exploration. The queue is not needed as the sweep-line method imposes a certain order of traversal (least progress first). We also see that the Sweep-line Exploration task takes an additional input value, namely Persistent initial states, which is a heuristic improving speed for certain systems that return to the initial state after executing. This is usually a very cheap way to speed up a large class of models, and is set to true by default. In order to better understand the method, a student may try setting the value to false and observe the performance on different models.

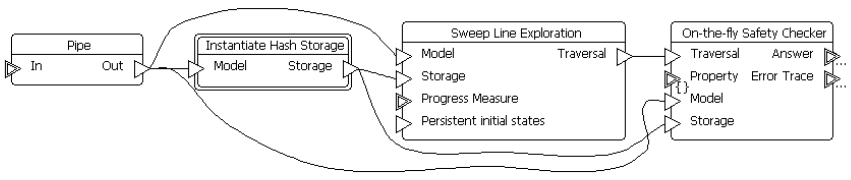


Fig. 8: Implementation of sweep-line safety checker.

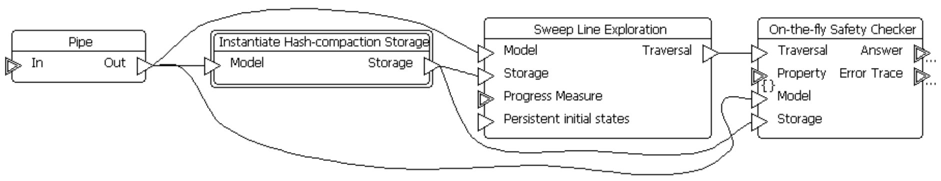


Fig. 9: Job for checking safety properties using a combination of the sweep-line method and hash compaction.

Finally, let us consider a researcher who experiments with the sweep-line method, knowing that a good way to obtain even better performance is to combine different methods. In addition to the sweep-line method, the researcher also knows about hash compaction, which, instead of storing the full states, just stores a hash value. This saves a lot of memory, as the hash value is usually 4 or 8 bytes whereas the full state is usually thousands of bytes. The caveat is that the method may not explore all states as several states may have the same hash value. The researcher thinks that using the sweep-line method to remove states known not to be encountered again may reduce this probability and use less memory than using either method by itself. The reasearcer therefore deletes the Instantiate Hash Storage macro, replaces it with a Instantiate Hash-compaction Storage macro, and obtains Fig. 9. If it is found that this method performs well, the constructed task can be saved and added to the tool so others can benefit in the future.

Suppose that the researcher wishes to test the performance of the new method and eliminate as much overhead to get as accurate results as possible. Looking at the details of the Hash-compaction Storage in Fig. 10, the researcher observes that it is possible to tune the initial size of the hash-table used to store states. Initially, this value is set rather small (1000) as the overhead in real cases is relatively small. If the researcher knows that the models uses for testing have at most 100,000 states and that the used computer has sufficient memory to handle this, the values can be set to 200,000 so the hash table never needs to be re-balanced, which will improve the performance slightly and factor out the cost of re-balancing a hash table, which provides a more fair and accurate comparison between the standard sweep-line method and the new method.

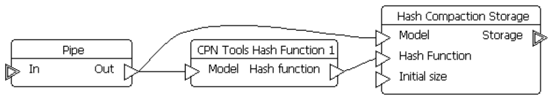


Fig. 10: Details of the hash compaction storage.

Finally, we may argue that even the task in Fig. 5 is not that simple as it contains eight tasks, of which only two or three are really interesting. We can instead create a new macro like the one in Fig. 11, where we have basically removed the inputs to model file and SML file from the task in Fig. 5 and exported the ports. We can go even simpler and remove the Property input port as well by replacing it with a set of standard properties that make sense for all models.

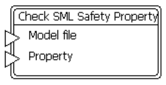


Fig. 11: More abstract top level of the safety checker from Figs. 3–5.

5 Conclusion and Future Work

In this paper we have proposed the JoSEL language for specification and execution of verification jobs in model checking. By means of an example we have demonstrated how JoSEL can be used to define a simple safety checker, which can be modified easily at different levels of abstraction. We have informally defined the semantics of JoSEL and indicated an efficient implementation in ASAP.

An important requirement for JoSEL was to support users at different experience levels and with different reasons to use verification tools. We find that JoSEL supports this because of the hierarchy concept, as is supported by the examples given in Sect. 4.

Future work includes considering extensions of the JoSEL language. One interesting extension is to allow cyclic connections, so that we can iteratively improve a verification result. This is interesting for implementing incremental improvement of approximative reduction techniques, e.g., by iteratively increasing the size of the table used for bit-state hashing [5] until we have refuted the property or a certain threshold has been reached. The reason for not just doing this is that it would make the implementation more complex and, more importantly, it would be easier for users to make mistakes, as Prop. 2 would no longer hold. It would also be interesting to add a way to terminate a task or a set of tasks. This has interesting applications to, e.g., state-of-the art model checking, in which we start model checking with all or at least several available model checking techniques. As soon as the first technique provides a definitive answer we terminate the execution of the other techniques. This should be fairly easy to do, and the main issue is to find a good graphical representation for this mechanism. A variant of macro tasks may be used as a composition mechanism stating which tasks to terminate. Many control structures need not be part of the language but can be implemented as tasks that do not correspond to anything in the underlying model-checking platform. On such example is an If task, which takes as input a boolean value test and an arbitrary value value. Depending on the value of the test, the value is either transmitted to a then or an else output port.

Another interesting direction is to investigate scheduling across multiple machines. It should be possible to extend JoSEL with annotations describing the time-wise and space-wise cost of a task and the produced values. It should then be possible to intelligently distribute data to the appropriate machines, taking into account that it may sometimes be more efficient to re-calculate data than to transmit results.

References

1. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*, chapter 12. The MIT Press, 1999.
2. Eclipse Graphical Modelling Framework (GMF). `www.eclipse.org/modeling/gmf/`.
3. Eclipse Modelling Framework (EMF). `www.eclipse.org/modeling/emf/`.
4. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE'01*, volume 197 of *IFIP Conference Proceedings*, pages 377–394. Kluwer, 2001.
5. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
6. L.M. Kristensen and M. Westergaard. The ASCoVeCo State Space Analysis Platform: Next Generation Tool Support for State Space Analysis. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 1–6, 2007.
7. T. Margaria, R. Nagel, and B. Steffen. Remote Integration and Coordination of Verification Tools in JEIT. In *Proc. of ECBS'05*, pages 431–436. IEEE Comp. Soc. Press, 2005.
8. R.S. Pressman. *Software Engineering*. McGraw-Hill, 1997.
9. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.
10. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
11. W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.