Journal of Grid Computing (2023) 21:39
https://doi.org/10.1007/s10723-023-09666-x

RESEARCH

Visual Low-Code Language for Orchestrating Large-Scale Distributed Computing

Kamil Rybiński · Michał Śmiałek · Agris Sostaks · Krzysztof Marek · Radosław Roszczyk · Marek Wdowiak

Received: 31 January 2023 / Accepted: 18 April 2023 / Published online: 4 July 2023

Abstract Distributed, large-scale computing is typically performed using textual general-purpose programming languages. This requires significant programming skills associated with the parallelisation and distribution of computations. In this paper, we present a visual (graphical) programming language called the Computation Application Language (CAL) to raise abstraction in distributed computing. CAL programs define computation workflows by visualising data flowing between computation units. The goal is to reduce the amount of traditional code needed and thus facilitate development even by non-professional programmers. The language follows the low-code paradigm, i.e. its implementation (the editor and the runtime system) is available online. We formalise the language by defining its syntax using a metamodel and specifying its semantics using a two-step approach. We define a translation of CAL into an intermediate language which is then defined using an operational approach. This formalisation was used to develop a programming and execution environment. The environment orchestrates computations by interpreting the intermediate language and managing the instantiation of computation modules using data tokens. We also present an explanatory case-study example that shows a practical application of the language.

Keywords Large scale computing · Low-code languages · Distributed computations · Formal language semantics

All author's are contributed equally to this work.

K. Rybiński · M. Śmiałek (✉) · K. Marek · R. Roszczyk · M. Wdowiak
Faculty of Electrical Engineering, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warszawa, Poland
e-mail: michal.smialek@pw.edu.pl

K. Rybiński
e-mail: kamil.rybinski@pw.edu.pl

K. Marek
e-mail: krzysztof.marek@pw.edu.pl

R. Roszczyk
e-mail: radoslaw.roszczyk@pw.edu.pl

M. Wdowiak
e-mail: marek.wdowiak@pw.edu.pl

A. Sostaks
Institute of Mathematics and Computer Science, University of Latvia, Riga, Latvia
e-mail: agris.sostaks@lumii.lv

1 Introduction

Over many years, textual general-purpose programming languages have dominated software development. Such languages have multiple advantages, allowing programmers to develop solutions for different domains in the same language. At the same time, programming in contemporary programming languages like Java, Python or C# still necessitates significant professional skills. Especially challenging is the construction of time-efficient computation software that uses parallel and distributed processing. Considering that experienced programmers are scarce on the market, the need for new programming approaches is constantly growing. These approaches should reduce complexity by raising the level of abstraction and removing unwanted technology-related issues. This way, they would be accessible to non-professional programmers or even to domain experts.

This tendency to reduce the complexity of programming and raise the abstraction at which programming constructs are formulated led to the emergence of the low-code approach [1]. Low-code solutions are predominantly based on creating visual, model-based languages [2] with the aim of making them more understandable and accessible. It can be argued that such a solution should be easier to use by inexperienced programmers and raises the productivity of programming [3]. For this reason, the usage of visual programming languages is recently gaining popularity in engineering, and education [4,5]. This can be observed especially in the field of distributed computing, such as IoT [6], which shares multiple similarities with more powerful distributed Large-Scale Computing platforms.

Typically, low-code systems are used to develop web-based business applications. However, recently it has been observed that the low-code paradigm can be easily applied to solve complex computation problems (using, e.g. Artificial Intelligence modules) [7]. This can be achieved by wrapping certain fragments of computation logic into computation units. These units can then be (re-)used when constructing computation applications at a significantly higher level of abstraction. This would lead to the emergence of a graphical (visual) programming language that would allow for expressing orchestrations (or choreographies [8]) of many computation units.

The main challenge for such a visual language would be dealing with typical computation parallelisation issues. Prominently, these issues pertain to High-Performance Computing (HPC) systems [9]. These systems focus on using powerful co-located homogeneous environments called supercomputers with strong bindings between computation nodes, thus allowing for strongly parallelised computations. Such approaches are mainly used by big research institutions and enterprises. This is due to the very high cost of operation and the requirement of expert knowledge of parallel computing to use the potential of such machines fully. Regular users would instead need an approach which we could call Large-Scale Computing (LSC). This approach would focus on using multiple distributed computation nodes, where each node is way less powerful than any supercomputer. However, linked together and parallelised, they can be used to solve advanced computation problems much faster than with the help of a single node. In both HPC and LSC, the main problem in parallelisation is the passing of data. However, the specific challenges differ. In LSC, nodes are distributed and connected through the Internet. Data transfer speeds are thus significantly slower than in HPC. HPC supercomputers are homogeneous, and their computation nodes are connected within a single location.

Considering the above, the main focus of the LSC approach is the management of data flow between computation nodes. The key is to minimise the overhead caused by data transfer and maximise computation speed through efficient distribution of workload between many nodes. In other words, computations should be controlled by the flow of data between the nodes where computations are performed. In such a data flow-driven approach to LSC, the entire computation is separated into steps. Each step produces results that become input data for the next steps until the final result is computed. Separating complex computations into steps allows treating each step as an independent computation unit placed in a separate container. Containerisation facilitates the orchestration of computations, as each step can follow one of the multiple solutions widely used in computation centres worldwide.

Moreover, the data-driven approach would allow for better utilisation of computer resources as the computation can start only when the input data is available. This reduces the amount of reserved but unused computation resources. Separation of computation steps into containers allows for easier reuse of already developed pieces of the application. The end-user would not need to edit the code of the computation step and treat it as a black box that can be connected to other computation steps to create new applications.

In this paper, we approach the creation of a graphical distributed computations language from the point of view of the low-code paradigm. It should be noted that several approaches to representing parallel and distributed computations in a graphical form already exist (see the next section). However, none of them seems to draw from the results of research on model-driven development that is used in constructing low-code languages. This includes aspects such as the notation's usability and the formalisation of its syntax and semantics. We thus present the Computation Application Language (CAL), developed as part of the BalticLSC Platform [10,11].

The paper contributes by introducing the precise syntax and semantics of CAL and the details of its implementation. The language's abstract syntax is defined through a metamodel with concrete syntax following best practices in this area [12]. We aimed to make the language accessible to non-professional programmers and domain experts according to the principles of low-code software development. Moreover, our work includes the definition of the language's semantics using translational and operational approaches. This allows for the unambiguous construction of execution environments for the language and illustrates a method for constructing similar scientific workflow environments.

In the next section, we provide related work referencing previous approaches to constructing visual computation languages. Section 3 provides a brief informal introduction to the presented language. This is followed by presenting CAL's syntax (abstract and concrete) in Section 4 and its semantics in Section 5. Section 6 shows how CAL's semantics was used to implement its execution environment on the web. This environment was used to conduct several case studies where one of them is presented in Section 7. We conclude with a discussion and proposition for the future development of CAL and its environment.

2 Related Work

As a recent study shows, the development of computation applications (HPC, parallel processing) is dominated by textual programming languages [13]. Only 5% of the developers use purely visual languages. The dominating model is to use a general-purpose textual programming language and equip it with features specific to the existing parallel and distributed programming models like MPI or OpenMP [14]. In these approaches, programmers need to deal with relatively low-level issues like message passing and memory sharing. This means that programmers need to have appropriate skills due to the significant technical complexity of parallel programming models.

The role of visual notations is to reduce the "accidental" complexity [15] and raise the level of abstraction at which programs are formulated through presenting computation flows in a graphical form. It is already a long-time discussion on whether visual notations are of benefit to professional programming [16]. However, they are more comprehensible for novice programmers [17] and increase the capability to create a mental representation of computation problems [18]. This generally stems from the fact that diagrams are most often better than text in expressing complex issues, including complex programs [19].

In the past, various graphical notations were used to assist parallel program development [20–22]. Thus, the idea to apply visual languages to high-performance computations (including parallel and distributed programming) has emerged quite early [23]. As a natural consequence, several visual programming systems have been proposed [24,25] together with graph grammars [26] and models [27] supporting the definition of parallelised computation. Syntactically, practically all such languages support graph structures, where graph nodes define the computation elements and graph arcs define data or control flows. This is also the case for the Computation Application Language (CAL) presented in the current work. However, CAL has several characteristics that distinguish it from other such languages.

The CODE, CODE 2.0, and Hence languages [28–30] are based on graph structures, where nodes define simple operations and the edges represent the order of their execution. Such a visual approach allows for a better representation of the concurrency of computations. These languages were used to define computations using atomic operations, and the graph was used to generate code directly executed on machines. Compared to them, CAL focuses on less granular computation steps encapsulated in separate containers. In CAL, graphs define sequences of container executions and the flow of data between these executions. This is different from the low-level code generation approach for which, over the years, many solutions were created. These solutions mainly focus on low-level parallelization of computations with the help of graphical languages, usually converted to C or C-like languages [31–34].

Another solution, extending the CODE language, is PEDS [35]. This tool allows for the visual mapping of computation units onto computation resources. It consists of four levels of abstraction: physical level, support level, visual language level, and application level. The PEDS tool allows for constructing applications with visual language and mapping their execution onto specific computation resources. In the PEDS visual language graphs, the nodes represent parallel processes, and the edges represent data dependencies between the processes. The use of PEDS requires its users to work within multiple layers of abstraction. Another similar tool was GRADE (Graphical Application Development Environment) [36], which consisted of a graphical editor to write parallel applications, a C code generator, and various distributed debugging tools. GRADE could perform computations on multiple nodes using only the graphically defined program.

A different parallel programming tool has been proposed by Delaitre et al. [37]. EDPEPPS is an IDE for visual parallel programs, providing a set of tools, including ones to build parallel programs, simulate their execution in heterogeneous environments and debug the programs executed in such environments. The main components of parallel computation are represented visually with the algorithms written textually as C-like procedures. Visual languages for parallel computation have also been developed to support cloud-based computations. The Visual Parallel Programming Environment [38] allows for the definition of low-level computation with a visual language, which is later translated into Java-MPI programs executed in the cloud. VPPE and other solutions [34,39] based on the generation of MPI programs require defining computation steps at an atomic level, which does not reduce the complexity required to develop a new solution. Compared to these approaches, CAL allows for defining computation with more complex steps and running it on a cloud without the need for the generation of low-level code.

Another use of a visual language for computation parallelisation has been proposed by Feng et al. [40] in the form of an extension of the Snap! language, designed to make learning parallel programming easier. This language has been inspired by MIT's Scratch project [41] that uses interlocking blocks instead of other already established visual notations such as Petri Nets [42] or Colored Petri Nets [43] as a way to visualise the control flow between computation steps. The previously sequential Snap! The tool has been extended, and the blocks can be converted to OpenMP code and executed on the machine of choice. Students could use the blocks to define parallelised computations using parallelisation blocks with operations such as MapReduce or parallel Map and ForEach functions. Researchers believe that by providing a visual language, they successfully lowered the learning curve for parallel computing compared to traditional C-based textual languages. The results emerging from the performed assessment seem to confirm this statement. The CAL has been created with the same goal. However, it uses boxes and arrows instead of interlocking blocks and operates on a less atomic level of computation steps.

A similar goal was the basis for the emergence of Scientific Workflow Management Systems [44]. Some of them, like Galaxy, Taverna, Kepler, and WS-PGRADE use graphical notations to denote the computation workflows. Seemingly, the most similar to our proposition is the WS-PGRADE system [45,46], which is the successor of the already mentioned GRADE system [36]. This system provides a web portal that allows for the creation and execution of computations in the form of workflows. What is important, it offers a visual workflow language where the execution of computation units (services) is controlled by the flow of data (files). Several solutions used in WS-PGRADE are close to those used in CAL and BalticLSC. Thus we will also refer to them in this paper's further text. Especially the data-flow-driven characteristics distinguish both solutions from the rest. It can be noted that WS-PGRADE was used to develop several dedicated portals for specific computation domains. In turn, BalticLSC aims to provide a common, user-friendly workspace where various domain-specific applications can easily be built using standard computation modules. Also, as mentioned in the introduction, CAL can be distinguished through its strict formalisation and usage of low-code (model-driven) approaches. It can be noted that both WS-PGRADE and BalticLSC use the orchestration approach with a central system that controls the flow of computations and data. It can be contrasted with the Flowbster system [47], which uses the choreography approach. In this system, workflows can be created as "autonomous graphs" of computation nodes that can be executed in the cloud. Moreover, it uses textual rather than visual notation.

The variety of approaches to representing scientific workflows calls for a common framework for easy-to-use, web-based workflow editors. An interesting attempt in this direction was that proposed by Gesing et al. [48]. They have introduced a generic data model and a design model for a workflow editor (Generic Web-based Workflow Editor – GeWWE). Based on this, they attempted to build a prototype graphical editor capable of generating workflow applications in different textual languages. Unfortunately, this attempt did not result in a fully developed system. In our approach with CAL, we propose a fully developed data model (metamodel) with semantics that allows for building a fully operational translator and execution engine.

The BalticLSC CAL editor allows for the web-based development of parallel applications. Similar web-based approaches to parallel computation can already be found, allowing for the graph-based definition of programs [49] or just providing a general gateway to distributed computation resources [50]. In a survey performed by Calegari et al. [51], researchers examined multiple existing web-based HPC solutions and defined general requirements for such platforms. The BalticLSC Platform fulfils these requirements and attempts to exceed them through, e.g. allowing for the visual definition of general-purpose computations with CAL. Many visual parallel computing languages on such platforms are domain specific as it simplifies the challenge [52], but simultaneously, it makes the solution less universal. Kubeflow [53] is a good example of such a domain-specific visual language, created to help develop machine learning pipelines running on Kubernetes [54]. In addition to allowing for the visual pipeline definition, Kubeflow allows for the execution of computations on standard Kubernetes clusters, allowing for easier management of computation resources. The BalticLSC Platform and CAL have similar foundations, with CAL being domain-independent and more uniform, allowing for its wider use while still utilising the easier orchestration of computations provided by containerisation and Kubernetes technology. To help with parallel application development, CAL has multiple skeletons similar to those described by Zandifar et al. [55] that help with the automatic parallelisation of computation similar to the MapReduce operations.

The specifics of the BalticLSC Network have many similarities with the volunteer computing platforms such as BOINC [56] or Seti@Home [57]. In both solutions, advanced computations are performed in parallel on multiple machines connected via the Internet. However, the BalticLSC Network mostly consists of comparatively larger resources (from small clusters to even HPC solutions) with the more stable (not voluntary) connection of nodes to the network.

3 CAL Overview

As a low-code language, CAL strives to be mostly self-explanatory. Thus, we will start introducing the language with a simple example. This should provide an intuitive understanding of the language constructs without studying a formal language specification.

Our example operates in the domain of video editing. The aim is to process black-and-white films with subtitles. Each film should be colourized, and its subtitles translated into a specific language. The subtitles should be appropriately mixed within the video file. What is important, we would like to process many such films in parallel.

As a typical representative of low-code languages, CAL is strongly based on graphical syntactic constructs. This is illustrated in Figs. 1 and 2 that contain the full application for our example problem. The first figure shows an elementary application for processing a single film ("VS Mixer" – video and subtitle mixer). It receives a video file ("Video Input") and a subtitle file ("Subtitle Input") and produces a processed film ("Output Film"). The application uses three computation modules. The first one ("Video Colorizer") performs automatic colourization. The second one translates subtitles ("Subtitle Translator"). As we can see, both of these modules can be run in parallel. Their results form the input to the third module ("Subtitle Mixer") that embeds the translated subtitles into the

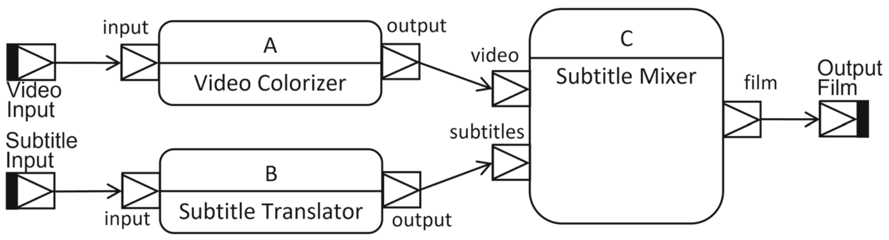


Fig. 1 Simple CAL application (VS Mixer)

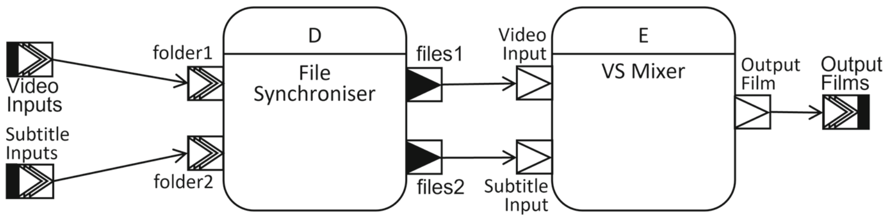


Fig. 2 Extended CAL application

final film file. Note that the execution of the application is controlled by the data that flow between module executions.

In the second figure (Fig. 2) we can see an application that enables the parallel processing of many films. We can notice that it uses the VS Mixer app (Fig. 1) as part of its code. On its input, it receives a folder of video files ("Video Inputs") and a folder of subtitle files ("Subtitle Inputs"). These folders are handled by the "File Synchroniser" module. This module creates pairs of files (for example, based on specific file naming rules) and sends these pairs sequentially to its outputs ("files1" and "files2"). Each of such pairs is then input to an instance of the "VS Mixer" application. The resulting films are placed by the mixer instances into a specified output folder ("Output Films"). One notation element that might be non-intuitive is the "data pin" symbols. Generally, single files are denoted by a single triangle symbol, folders are denoted by a triple triangle symbol, and file/folder sequences are denoted by a black triangle symbol. This will be explained in more detail in the following section.

Note that the computation modules ("Video Colorizer", "Subtitle Translator" etc.) form the elementary building blocks of the language. In this sense, the language is extendable through defining new computation modules and thus extending their libraries. It is up to the language users to define their modules or to use the existing ones found in the library. The role of the language is to provide means for the parallelisation and distribution of computations. The language runtime system takes care of running instances of appropriate computation modules on appropriate computation resources and transmitting data between these instances. Thus, an important part of the runtime system is a component that performs computation job brokerage (assignment of jobs to specific computation nodes).

It is worth noting that some of the CAL constructs influence the way module execution is parallelised. For example, let us consider the file pairs produced by the "File Synchroniser" (see Fig. 2). The creation of such pairs early in the processing facilitates the optimisation of job brokerage. In this case, both the colourization and subtitle translation tasks can be assigned to the same computation node, thus avoiding potentially costly file transfers.

In a practical implementation of our language, computation modules are provided as container images. Each computation node operates an instance of a container management system (like Kubernetes or Docker Swarm). The CAL runtime engine then controls the distribution of container instances to appropriate container management system instances.

4 Language Syntax

To define the CAL syntax, we use typical techniques of software language engineering for defining graph-based languages [58]. This consists in defining the language's abstract syntax (internal structure of language constructs) and concrete syntax (visual representations of language constructs as seen by the language users). The abstract syntax is defined using a metamodel which is a typical approach. The concrete syntax is defined through some examples and an informal description of how visual representations can be used.

Basically, CAL consists of just a few constructs - unit calls, data pins (declared and computed), and data flows. The respective metamodel can be seen in Fig. 3, but a detailed description of the main classes and concrete syntax follows in Tables 1, 2, 3, and the next paragraphs.

CAL meta-model is centred around two main meta-classes - ComputationUnit and ComputationUnitRelease. The first of them is used to distinguish the logical units of computation that perform a specific function. The second of them represents particular versions of those units tied to given implementations. Both of those meta-classes are further described by their respective

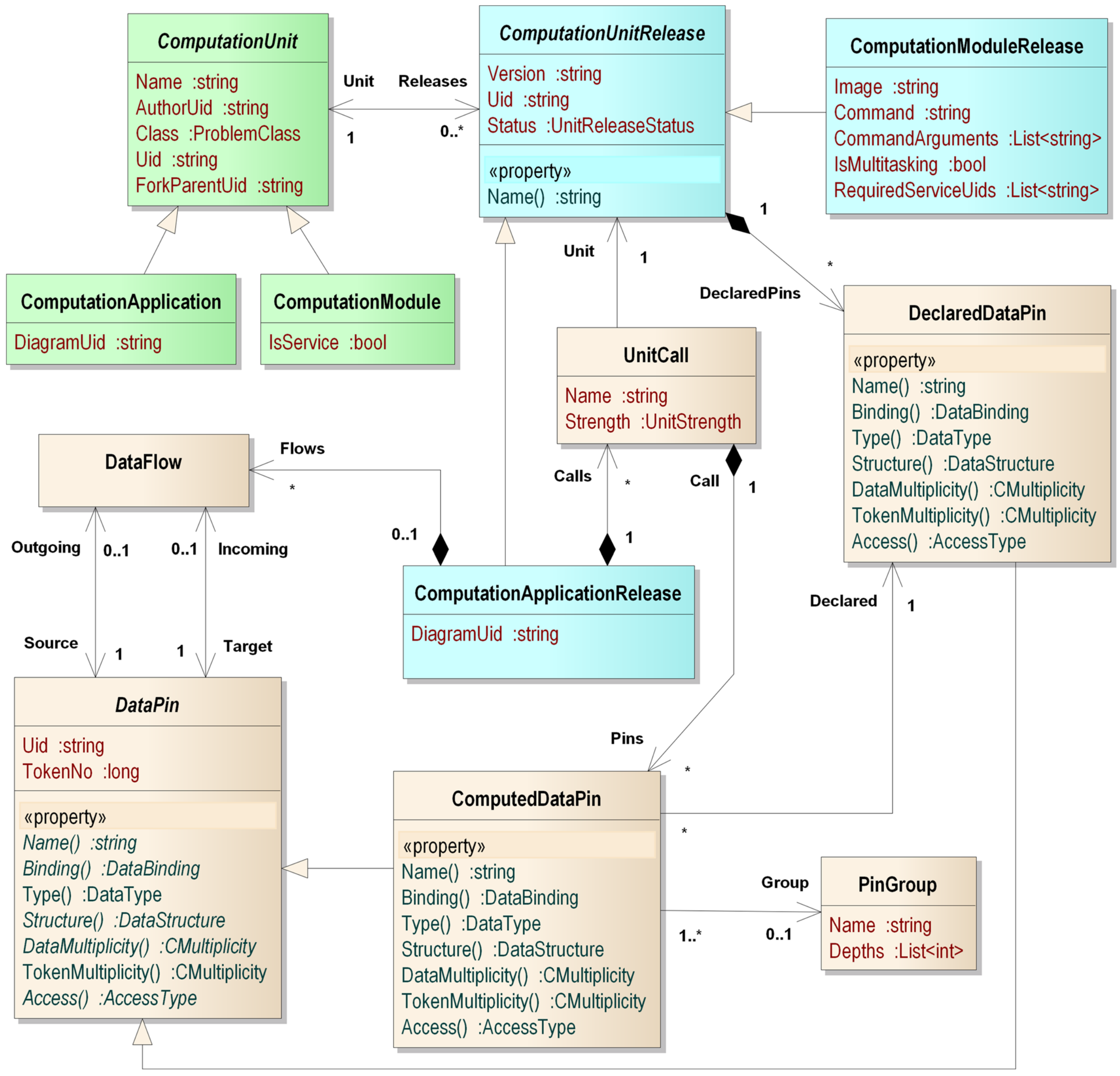
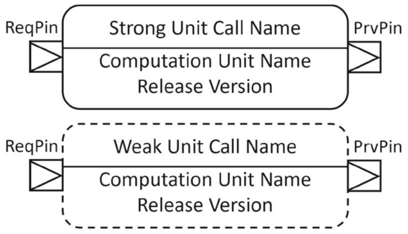


Fig. 3 CAL abstract syntax

descriptors. Additionally, both of them are also further divided according to the exact abstraction level on which they occur.

ComputationModules and ComputationModuleReleases represent the bottom layer of units and releases, corresponding to atomic parts of computations executed on particular computation nodes. ComputationApplications and ComputationApplicationReleases represent elements situated higher in a hierarchy and typically encompass the functionality of many smaller units. Specific dependency between units is defined at the level of their releases, using the UnitCall meta-class. Instances of this meta-class are contained in ComputationApplicationReleases and point to UnitReleases invoked by them. The exact sequence of unit calls in the application is defined using representatives of the DataFlow meta-class specifying the acceptable paths that data can flow between the called units. These data flows connect DataPins observing their type. These data pins represent data input or output from respective computation units. Data pins associated with unit calls (computed pins) refer to data pins that are associated

Table 1 Abstract and concrete syntax of unit calls

Unit Call	
Abstract Syntax	Concrete Syntax
The Unit Call class represents the invocation of computation units within application workflows, thus being the main syntactic element of every CAL program. Unit calls have data pins which denote data sets received as inputs and outputs of the computation. **Attributes and Associations.** Name: string - the name of the unit call. Strength: UnitStrength - unit calls might be Weak or Strong. Strong unit calls require all underlying computations to be computed within a single computation node. Weak unit calls do not set any restrictions on assignment to nodes. Unit [1] - computation unit release which is invoked by the unit call. Pins [*] - computed data pins owned by the unit call. They refer to the data pins declared by the invoked unit release and can be configured according to the needs of the current application.	Unit calls are depicted as rounded rectangles. The upper part of the unit call contains the unit call's name, and the lower part contains the name of the invoked computation unit release which consists of the unit's name and release version. Data pins owned by the unit call are typically placed on the left and right of the unit call (required on the left, provided on the right). The strength of the unit call is denoted by the outline shape of the rectangle - strong unit calls have solid lines and weak ones are dashed.  ReqPin Strong Unit Call Name PrvPin Computation Unit Name Release Version ReqPin Weak Unit Call Name PrvPin Computation Unit Name Release Version

with unit releases (declared pins). In other words, each ComputedDataPin points to a compatible DeclaredDataPin contained in the respective ComputationUnitRelease. These DeclaredDataPins specify details about data received and produced by unit releases, and in the case of applications, also serve as starting and finishing points for data flows.

Computation units and their releases do not have concrete syntax and are not presented in CAL specifications. Other language syntactic constructs have appropriate graphical representations as presented in Tables 1, 2, 3. Each of the tables briefly explains the abstract syntax (the metamodel elements in Fig. 3) and the concrete syntax through appropriate examples.

Considering the configuration and token multiplicity of input and output pins, we can distinguish several reference module types. These types can be combined into hybrid types (e.g. splitter-joiner).

- Simple processor (one single input, one single output) – a most common computation unit that performs some algorithm on input data, creating output data.
- Data separator (one single input, many single outputs) – a computation unit that splits the input data

Table 2 Abstract and concrete syntax of data flows

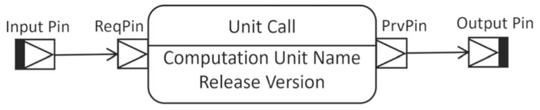
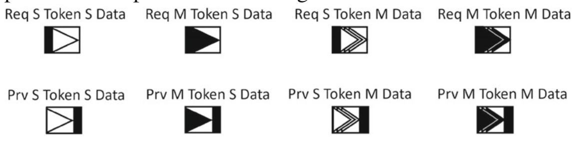
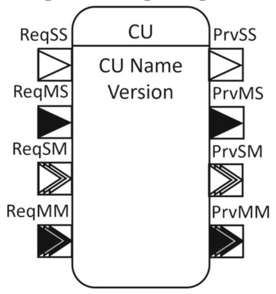
Data Flow	
Abstract Syntax	Concrete Syntax
The Data Flow class represents the transition of data tokens between data pins. Every data flow connects exactly two data pins. The flow of data is from a declared required or computed provided pin to a declared provided or computed required pin. There can be just one outgoing data flow from any CAL element, but multiple incoming data flows are possible. **Associations.** Target [1] - the data pin that consumes data tokens. Source [1] - the data pin that produces data tokens.	Data flows are depicted as arrows. They must connect two data pins. The figure below is a simple example of a CAL program with two data flows (arrows) appropriately connecting two declared pins with two computed pins.  Input Pin ReqPin Unit Call PrvPin Output Pin Computation Unit Name Release Version

Table 3 Abstract and concrete syntax of data pins

Data Pins	
Abstract Syntax	Concrete Syntax
The Data Pin class is an abstraction that encompasses declared and computed data pins. Data pins represent the inputs and outputs of computation units (applications and modules).This refers to specific data sets required and provided by computation unit releases. Each data pin is characterised by its data and token multiplicities, its data type (e.g., JSON, XML, Image, etc. ...), its metadata structure (for structured data types), and its access type (e.g., MongoDB, FTP). The Declared Data Pin class represents the inputs and outputs of applications or modules. In turn, the Computed Data Pin class represents instantiations of declared data pins that are parts of unit calls. Computed pins are derived from the declarations of the respective computation unit releases that are invoked by the including unit calls. More specifically, they are derived from the respective declared pins. Data pins can be connected by data flows being their respective sources and targets. A required declared pin can only be a source, while a provided declared data pin can only be a target. Computed data pins act in the opposite way - required computed pins can be targets while provided computed pins can be sources. **Key attributes and associations.** Name: string - specifies the name of the data pin. Binding: DataBinding - specifies the data binding for the data pin. Binding might be RequiredStrong, RequiredWeak, Provided or ProvidedExternal. Required data pins define data sets that are consumed, and provided data pins define data sets that are produced. Strong data pins define mandatory data sets, and weak data pins define optional data sets. DataMultiplicity: CMultiplicity - specifies the multiplicity of data items in the data set defined by a single token. It might be Single (e.g. a single file) or Multiple (e.g. a folder of files). TokenMultiplicity: CMultiplicity - specifies the multiplicity of tokens consumed or produced by the data pin. It might be Single (one token produced) or Multiple (multiple tokens produced).	**Declared Data Pin** Declared Pins for applications are defined within CAL programs using the notation of rectangle. The shape of the rectangle is determined by the values of the pin's attributes. The upper part of the figure below contains the required declared data pins, while the lower part contains provided declared data pins of an application. They are distinguished by the placement of a black bar - a required data pin has it on the left side, and a provided data pin has it on the right side.  Req S Token S Data Req M Token S Data Req S Token M Data Req M Token M Data Prv S Token S Data Prv M Token S Data Prv S Token M Data Prv M Token M Data Data multiplicity is depicted by the number of triangles - "single" is denoted by one triangle, and "multiple" is denoted by three. Token multiplicity is depicted by the colour of the triangles - "single" has a white interior, and "multiple" has a dark interior. The name of the declared data pin is placed near (e.g. above) the rectangle. **Computed Data Pin** Computed pins are depicted as rectangles pinned to unit call boxes. The figure below shows examples, where pin names denote: Req - required, Prv - provided; two last letters provide the token and data multiplicity accordingly, S - single, M - multiple. Required pins are on the left, provided on the right side of the unit call. Multiplicities of computed data pins are denoted as for the declared data pins. The name of the computed data pin is placed near (e.g. above) the rectangle.  CU CU Name Version ReqSS ReqMS ReqSM ReqMM PrvSS PrvMS PrvSM PrvMM

into two or more output data sets that will be further processed using different means; separation can be done using some computation algorithm.

- Data splitter (one single input, at least one multiple outputs) – a computation unit that splits a data set into smaller parts that will be further processed in the same way; splitting can be done using some computation algorithm.
- Data joiner (many single inputs, one single output) – a computation unit that joins several data sets (typically resulting from different processing paths) into a single data set; joining can be done using some computation algorithm.
- Data merger (at least one multiple input, one single output) – a computation unit that merges several data sets of the same type into a combined data set; merging can be done using some computation algorithm.

It can be noticed that this classification of nodes is somewhat similar to that found in the WS-PGRADE system [45] (cf. Collector and Generator nodes). However, in our approach, the exact categories and semantics slightly differ.

5 Language Formal Semantics

Being a low-code programming language, CAL needs a precise definition of its semantics to be used during runtime. To define it, we use a hybrid approach consisting of two steps. In the first step, we use the translational semantics approach [58] (see Chapter 10). For this purpose, we define an intermediate language called CAL-Executable. Based on this, we specify a set of translation rules that map CAL constructs onto the constructs of CAL-Executable. In the second step, we use the operational semantics approach [59] (see Chapter 8). For this purpose, we define an abstract machine with a set of transitions defining its behaviour. This machine defines the execution of CAL-Executable programs.

The reason for this hybrid approach lies in the characteristics of CAL. The language is graph-based and thus it is not trivial to define operational semantics directly. At the same time, it is not possible to use translational semantics alone. This is due to special requirements for the execution of CAL programs (parallelisation and distribution of computations through container instances). This prevents us from using a standard existing language (with known semantics) as the target for the translation.

5.1 CAL-Executable Definition

Before specifying CAL semantics formally, we first need to define the CAL-Executable syntax. We do it in the same way as for the CAL syntax - through metamodel, as shown in Fig. 4. The metamodel is based on three main classes CTask, CJobBatch, and CJob. CTask represents the whole computation task solving a specific problem. CJob represents the smallest portion of a computation task, connected to a particular code run in a container. CJobBach represents a strongly dependent set of CJobs that need to be run on the same cluster. An additional class, CService represents containers like databases that need to be running constantly and are required by certain CJobs.

CJobs and CServices are the elementary executable elements contained in CJobBatches and specialise in a more general CJobBatchElement metaclass. This metaclass is used to group common features of CJobs and CServices, like paths to particular container images. Even more general is the CExecutable metaclass which is specialised for all metaclasses that represent executable elements, including CTasks and CJobBatches. It is used to provide the identification of an executable element within the runtime environment. Moreover, every CExecutable instance can contain many CDataToken elements. The CDataToken metaclass represents the metadata of the data elements (e.g. files) passed between the executable elements.

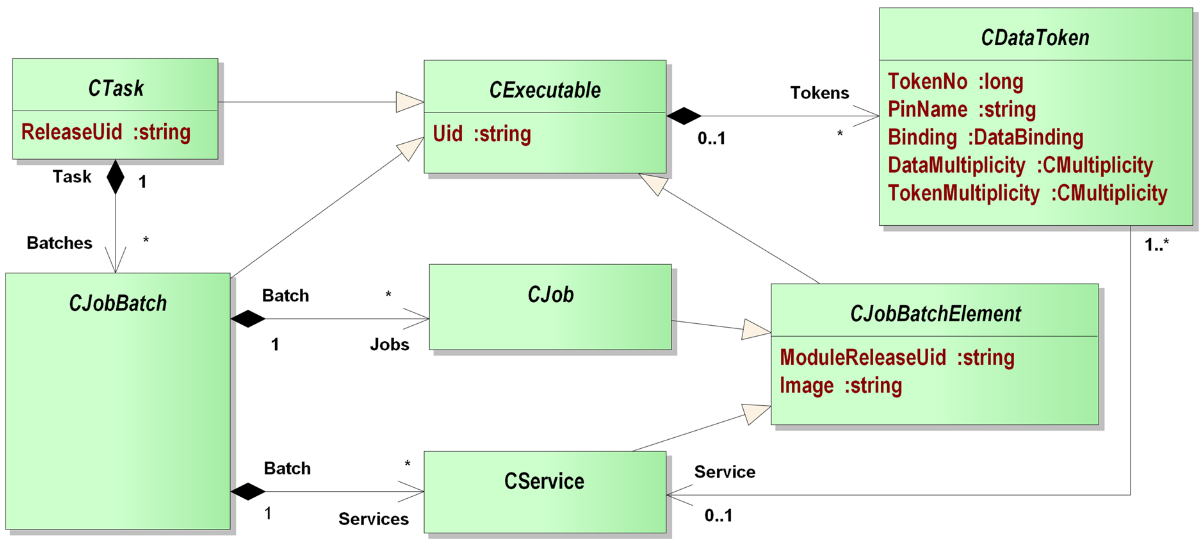


Fig. 4 CAL-Executable abstract syntax

Figures 5, 6 and 7 contain examples of CAL-Executable syntax. As can be noticed, the syntax is textual. Moreover, each program can be expressed in a linear form (tasks containing batches and batches containing jobs).

Figure 5 shows a translation of the VS Mixer program (see Fig. 1) into CAL-Executable. As we can see, the translated program contains one task with one embedded batch. The batch contains three jobs corresponding to the three unit calls of the source VS Mixer program. Each job contains data token definitions corresponding to its required and provided pins. The batch contains data tokens corresponding to the declared pins of the overall application, where such a situation occurs when just a single batch is created in a CAL-Executable program.

In the runtime environment, this program is executed through the exchange of data token instances. Every such instance represents a particular piece of data (e.g., a file) to be processed by computation module instances. The initial data tokens are created based on the user input. Here, for instance, the application user should provide appropriate metadata that points to the files containing "Video Input" and "Source Input". This will cause the creation of appropriate two data token instances with respective token numbers (no=1 and no=2). This in turn will cause the initiation of a new task and its only job batch instance. This is because we have two "required strong" data tokens in the definition of the job batch that have matching token numbers.

The initiation of the new batch instance is followed by the initiation of contained job instances. Specifically, an instance of the Video Coloriser and an instance of the Subtitle Translator are created. This is due to that these jobs have "required strong" tokens where their numbers correspond to the already received two data token instances (no=1 for the Video Coloriser and no=2 for the subtitle Translator). When these two job instances finish execution, they produce appropriate data token instances (no=4 and no=5, respectively). This causes the initiation of an instance of the Subtitle Mixer module. Finally, this instance produces a token (no=3) which corresponds to the "provided" data token of the containing batch. This causes the finalisation of the batch instance and the whole program.

Figure 6 shows a translation of the extended CAL application (see Fig. 2. Note that this extended application calls the VS Mixer app. The translation was made in a "strong" mode, which means that all jobs should be contained in a single job batch. This results in more optimal data transfer but can negatively impact parallelisation (all job instances are executed in the same computation node).

The CAL-Executable program in Fig. 6 will be executed similarly to the program in Fig. 5 but with an additional job (File synchroniser). A significant difference is caused by the "multiple" data tokens in the job batch and File Synchroniser definitions. The tokens required by the job batch (and the File Synchroniser) have a "data multiplicity" set, which means that the batch expects to receive two tokens (no=1 and no=2) pointing to appropriate data sets (e.g. file folders). The tokens provided by the File synchroniser have a "token multiplicity" set. This means that they produce many tokens of each type (no=4 and no=5). As a result, the remaining jobs will have many instances, depending on

Fig. 5 CAL-Executable program for the simple CAL application

```
Task uid=t001 ruid=r001
  JobBatch uid=b001
    DataToken RequiredStrong no=1 pin="Video Input" data=Single token=Single
    DataToken RequiredStrong no=2 pin="Subtitle Input" data=Single token=Single
    DataToken Provided no=3 pin="Output Film" data=Single token=Single
    Job uid=j001 ruid=r101 image="blsc/vc"          →Video Colorizer
      DataToken RequiredStrong no=1 pin="input" data=Single token=Single
      DataToken Provided no=4 pin="output" data=Single token=Single
    Job uid=j002 ruid=r102 image="blsc/st"          →Subtitle Translator
      DataToken RequiredStrong no=2 pin="input" data=Single token=Single
      DataToken Provided no=5 pin="output" data=Single token=Single
    Job uid=j003 ruid=103 image="blsc/sm"           →Subtitle Mixer
      DataToken RequiredStrong no=4 pin="video" data=Single token=Single
      DataToken RequiredStrong no=5 pin="subtitles" data=Single token=Single
      DataToken Provided no=3 pin="film" data=Single token=Single
```

```
Task uid=t002 ruid=r002
  JobBatch uid=b002
    DataToken RequiredStrong  no=1 pin="Video Inputs" data=Multiple token=Single
    DataToken RequiredStrong no=2 pin="Subtitle Inputs" data=Multiple token=Single
    DataToken Provided no=3 pin="Output Films" data=Multiple token=Single
    Job uid=j004 ruid=r104 image="blsc/fs"          → File Synchroniser
      DataToken RequiredStrong no=1 pin="folder1" data=Multiple token=Single
      DataToken RequiredStrong no=2 pin="folder2" data=Multiple token=Single
      DataToken Provided no=4 pin="files1" data=Single token=Multiple
      DataToken Provided no=5 pin="files2" data=Single token=Multiple
    Job uid=j005 ruid=r101 image="blsc/vc"          →Video Colorizer
      DataToken RequiredStrong no=4 pin="input" data=Single token=Single
      DataToken Provided no=6 pin="output" data=Single token=Single
    Job uid=j006 ruid=r102 image="blsc/st"          →Subtitle Translator
      DataToken RequiredStrong no=5 pin="input" data=Single token=Single
      DataToken Provided no=7 pin="output" data=Single token=Single
    Job uid=j007 ruid=r103 image="blsc/sm"          →Subtitle Mixer
      DataToken RequiredStrong no=6 pin="video" data=Single token=Single
      DataToken RequiredStrong no=7 pin="subtitles" data=Single token=Single
      DataToken Provided no=3 pin="film" data=Single token=Single
```

Fig. 6 CAL-Executable program for the extended CAL application (strong)

```
Task uid=t003 ruid=r002
  DataToken RequiredStrong  no=1 pin="Video Inputs" data=Multiple token=Single
  DataToken RequiredStrong no=2 pin="Subtitle Inputs" data=Multiple token=Single
  DataToken Provided no=3 pin="Output Films" data=Multiple token=Single
  JobBatch uid=b003
    DataToken RequiredStrong  no=1 pin="Video Inputs" data=Multiple token=Single
    DataToken RequiredStrong no=2 pin="Subtitle Inputs" data=Multiple token=Single
    DataToken Provided no=4 pin="files1" data=Single token=Multiple
    DataToken Provided no=5 pin="files2" data=Single token=Multiple
    Job uid=j008 ruid=r104 image="blsc/fs"          → File Synchroniser
      DataToken RequiredStrong no=1 pin="folder1" data=Multiple token=Single
      DataToken RequiredStrong no=2 pin="folder2" data=Multiple token=Single
      DataToken Provided no=4 pin="files1" data=Single token=Multiple
      DataToken Provided no=5 pin="files2" data=Single token=Multiple
  JobBatch uid=b004
    DataToken RequiredStrong  no=4 pin="Video Input" data=Single token=Single
    DataToken RequiredStrong no=5 pin="Subtitle Input" data=Single token=Single
    DataToken Provided no=3 pin="Output Film" data=Multiple token=Single
    Job uid=j009 ruid=r101 image="blsc/vc"          →Video Colorizer
      DataToken RequiredStrong no=4 pin="input" data=Single token=Single
      DataToken Provided no=6 pin="output" data=Single token=Single
    Job uid=j010 ruid=r102 image="blsc/st"          →Subtitle Translator
      DataToken RequiredStrong no=5 pin="input" data=Single token=Single
      DataToken Provided no=7 pin="output" data=Single token=Single
    Job uid=j011 ruid=r103 image="blsc/sm"          →Subtitle Mixer
      DataToken RequiredStrong no=6 pin="video" data=Single token=Single
      DataToken RequiredStrong no=7 pin="subtitles" data=Single token=Single
      DataToken Provided no=3 pin="film" data=Single token=Single
```

Fig. 7 CAL-Executable program for the extended CAL application (weak)

the number of appropriate tokens (cf. number of files in the folders).

Figure 7 also shows a translation of the extended CAL application in the “weak” mode. This means that jobs can be distributed between several job batches. This can result in better parallelisation but can also impede data transfer times. Note that marking of applications and module calls as “weak” and “strong” indicates the sensitivity of computations on data transfer and influences division into computation batches. This is a unique characteristic of CAL which distinguishes it from previous such languages.

In our example, the CAL-Executable program is divided into two batches. Each of the batches contains its own set of data tokens. Moreover, the whole task contains a set of data tokens. This is due to that the initiation of the task is not equivalent to the initiation of one of its batches. The task will be initiated when token instances with no=1 and no=2 arrive. This will also initiate the batch with uid=b003. The other job batch will be initiated only after an instance of the File Synchroniser produces token instances with no=4 and no=5. Note that this time, multiple tokens produced by the instance of batch uid=b003 will cause the initiation of many instances of batch uid=b004.

5.2 Translation from CAL to CAL-Executable

Having defined the syntax of CAL-Executable, we can now start defining the semantics of CAL. Here, we provide the first part of the formal specification using a translational approach. We start by defining a utility function that is used within the translational rules to shorten them. Note that within the rules we refer to class and attribute names from the metamodel (see Fig. 3).

Definition 1 The function “child” is defined as follows:
$child(x : UnitCall, y : UnitCall) \longrightarrow y.Unit.Calls \ni x \vee \exists(z : UnitCall)$, such that $(y.Unit.Calls \ni z \wedge child(x, z))$

The “child” function is boolean and has two parameters - unit calls. It is true if the first unit call is (recursively) contained within a computation application release called by the second unit call.

With this definition, we define 7 translation rules. Each rule is presented in a uniform notation. The rule definition starts with a brief, informal textual description. This is followed by three sections. The “source” section lists and names a set of objects subject to the respective translation. The “condition” section defines the specific configurations of the objects listed in the “source” section. These configurations need to be fulfilled in order for the rule to be applied to these objects. Moreover, the rule will be applied to all configurations that fulfil the condition. The “target” section defines objects and their configurations that should be created as a result of applying the given rule.

1. Create a task for the outermost application release.
   Source:$car : ComputationApplicationRelease$, $uc : UnitCall$
   Condition:$uc.Unit = car \wedge \nexists(uc2 : UnitCall)$, such that $child(uc, uc2)$
   Target: $ct : CTask$, where $(ct.ReleaseUid = car.Uid)$

2. Create a batch for all application releases that are started/called in “strong” mode and that aren’t contained (directly or indirectly) in another application release started/called in “strong” mode.
   Source: $uc : UnitCall$, $car : ComputationApplicationRelease$, $ct : CTask$
   Condition: $uc.Unit = car \wedge uc.Strength = Strong \wedge \nexists(uc2 : UnitCall)$, such that $(uc2.Strength = Strong \wedge child(uc, uc2))$
   Target: $cjb : CJobBatch$, where $(cjb.Task = ct)$

3. Append the batch from rule (2) with jobs based on calls to module releases contained in the application release from rule (2) and in all application releases called by it (directly or indirectly).
   Source: $cmr : ComputationModuleRelease$, $cjb : CJobatch$
   Condition: $\exists(uc : UnitCall, car : ComputationApplicationRelease, uc2 : UnitCall, ct : CTask)$, such that $Rule(2)(uc, car, ct \rightarrow cjb) \wedge uc2.Unit = cmr \wedge child(uc2, uc)$
   Target: $cj : CJob$, where $(cj.Batch = cjb \wedge cj.Image = cmr.Image)$

4. Create separate batches for the rest of the calls to module releases, and add jobs based on these calls for each of them.

Source: $cmr : ComputationModuleRelease$, $ct : CTask$
Condition: $\exists(uc : UnitCall,\ car : ComputationApplicationRelease,\ uc2 : UnitCall)$, such that $(uc2.Unit = cmr \wedge child(uc2, uc)) \wedge \nexists(cjb : CJobBatch)$, such that $Rule(2)(uc, car, ct \rightarrow cjb)$
Target: $cjb : CJobBatch$, where $(cjb.Task = ct)$, $cj : CJob$, where $(cj.Batch = cjb \wedge cj.Image = cmr.Image)$

5. To each job, add tokens based on pins of the module related to it (contained in a call from which the job was created).
Source: $cj : CJob,\ cmr : ComputationModuleRelease,\ dp : DataPin$
Condition: $(\exists(cjb : CJobBatch)$, such that $Rule(3)(cmr, cjb \rightarrow cj) \vee \exists(ct : CTask, cjb : CJobBatch)$, such that $(Rule(4)(cmr, ct \rightarrow cjb, cj)) \wedge cmr.DeclaredPins \ni dp)$
Target:$cdt : CDataToken$, where $(cdt.PinName = dp.Name \wedge cdt.Binding = dp.Binding \wedge cdt.DataMultiplicity = dp.DataMultiplicity \wedge cdt.TokenMultiplicity = dp.TokenMultiplicity \wedge cj.Tokens \ni cdt)$

6. To each batch, add tokens based on the pins of the application or module release that was the basis for its creation.
Source: $cjb : CJobBatch,\ uc : UnitCall, cdp : ComputedDataPin$
Condition: $(\exists(car : ComputationApplicationRelease, ct : CTask)$, such that $(Rule(2)(uc, car, ct \rightarrow cjb)) \vee \exists(cmr : ComputationModuleRelease, ct : CTask)$, such that $(Rule(4)(cmr, ct \rightarrow cjb) \wedge uc.Unit = cmr) \wedge uc.Pins \ni cdp$
Target:$cdt : CDataToken$, where $(cdt.PinName = cdp.Name \wedge cdt.Binding = cdp.Binding \wedge cdt.DataMultiplicity = cdp.DataMultiplicity \wedge TokenMultiplicity = cdp.TokenMultiplicity \wedge cjb.Tokens \ni cdt)$

7. If the task contains more than one batch, add tokens based on all declared pins in the outermost application to that task.
Source: $ct : CTask,\ car : ComputationApplicationRelease,\ uc : UnitCall,\ ddp : DeclaredDataPin$
Condition: $\exists(cjb1 : CJobBatch, cjb2 : CJobBatch)$, such that $(cjb1 \neq cjb2) \wedge uc.Unit = car \wedge \nexists(uc2 : UnitCall)$, such that $(child(uc, uc2)) \wedge car.DeclaredPins \ni ddp$
Target:$cdt : CDataToken$, where $(cdt.PinName = ddp.Name \wedge cdt.Binding = ddp.Binding \wedge cdt.DataMultiplicity = ddp.DataMultiplicity \wedge TokenMultiplicity = ddp.TokenMultiplicity \wedge ct.Tokens \ni cdt)$

Note that the first four above rules are responsible for creating the structure of the target CAL-Executable program – the task with contained batches and jobs. The last three rules are responsible for creating tokens associated with appropriate tasks, batches, and jobs.

5.3 Operational Semantics of CAL-Executable

Having defined the translation from CAL to CAL-Executable we should now complement the specification of CAL semantics by formally defining the semantics of CAL-Executable. To do this, we use the operational semantics approach. We will define an abstract machine with a set of configurations and a set of transition relations (a transition system [60]).

Definition 2 A *CAL-Executable Abstract Machine* is a tuple $M = \langle T, B, J, \rho_B, \rho_J \rangle$ where:

- T is a finite set (of tokens)
- B is a finite set (of batches)
- J is a finite set (of jobs)
- $\rho_B : \overline{T} \longrightarrow B$ (is the batch execution starting relation)
- $\rho_J : \overline{T}, B \longrightarrow J$ (is the job execution starting relation)

where $\overline{X}$ denotes a finite set of elements where each element belongs to X.

These elements of the abstract machine correspond to the syntactical structure of CAL-Executable programs, consisting of job batches, jobs, and tokens. Batch and job sets correspond directly to the batch and job definitions in a CAL-Executable program. The token set is treated differently. For constructing token sets, we treat CDataTokens with the same TokenNo and Binding as the same token (even if they have different PinNames). Finally, the execution starting relations correspond to the containment of tokens in appropriate batches and jobs. For these relations, the tokens are also treated the same as described above.

Based on this definition, we can specify a transition system. First, we define the following instance sets:

- I_S is a finite set of unique identifiers
- $Ti = \{\langle t, i \rangle \mid t \in T,\ i \in I_S\}$ (is a finite set of token instances)
- $Bi = \{\langle b, i \rangle \mid b \in B,\ i \in I_S\}$ (is a finite set of batch executions)
- $Ji = \{\langle j, bi \rangle \mid j \in J,\ bi \in Bi\}$ (is a finite set of job executions)

According to this, our abstract machine during execution operates on appropriate sets of token, batch, and job instances. Token and batch instances are distinguished through unique identifiers. Job instances are identified through their assignment to specific batch instances.

The resulting transition system is thus defined as follows. Its set of configurations Γ is:

- $\Gamma = \overline{Ti} \times \overline{Bi} \times \overline{Ji}$

and the set of transition relations $\vdash$ is:

- $\vdash_B = \{\langle\langle at, b, j\rangle, \langle at, db, j\rangle\rangle \mid a, t \in \overline{Ti},\ d \in Bi,\ b \in \overline{Bi},\ j \in \overline{Ji},\ d \notin b,\ d_1 = \rho_B(a_1),\ \forall e \in a : d_2 = e_2\}$
- $\vdash_J = \{\langle\langle at, db, j\rangle, \langle t, db, cj\rangle\rangle \mid a, t \in \overline{Ti},\ d \in Bi,\ b \in \overline{Bi},\ c \in Ji,\ j \in \overline{Ji},\ c_2 = d,\ c_1 = \rho_J(a_1, d_1),\ \forall e \in a : e_2 = d_2\}$
- $\vdash = \vdash_B \cup \vdash_J$

The first transition set $\vdash_B$ pertains to creating new batch executions based on the batch execution starting relation ρ_B. The source configuration $\langle at, b, j\rangle$ is transformed such that a new batch execution d is added to the current set of batch executions. For such a transformation to be executed, two conditions need to be met. The first condition simply requires that the current set of batch executions b does not yet contain the new batch execution d. The second condition is applied to the current set of token instances. This set should contain a subset of token instances a, compliant with the batch execution starting relation ρ_B. By this, we mean that there exists a batch in relation ρ_B with exactly such a set of tokens, that all of these tokens are the first elements of token instance tuples in a and all of them have the same identifier (second element of token instance tuples). Moreover, the first element of the new batch execution tuple d_1, will be set to this above-mentioned batch, and the second element of the tuple d_2, will be set to the above-mentioned identifier.

The second transition set $\vdash_J$ pertains to creating new job executions based on the job execution starting relation ρ_J. The source configuration $\langle at, db, j\rangle$ is transformed such that a new job execution c is added to the current set of job executions. Moreover, a subset of token instances is removed from the current set of token instances. For such a transformation to be executed, two conditions need to be met. The first condition simply requires that the current set of batch executions contains a batch execution d that is the second element of the new job execution tuple c_2. The second condition is applied to the current set of token instances and is analogous to the second condition of the transition set $\vdash_B$, but pertaining to job execution. This also involves the batch d_1 (the first element of the batch execution tuple d) that needs to participate additionally in the job execution starting relation ρ_J.

6 Language Implementation

The presented syntax and semantics of CAL and CAL-Executable were used and implemented as a basis for constructing a full Large Scale Computing system - the BalticLSC system.[1] The system offers a web-based user interface and is currently freely available for application developers. The overall architecture of the system is presented in Figs. 8 and 14. Figure 8 shows the main components of the Master Node that are responsible for the management and execution of CAL programs. This includes mechanisms for distributing computations to be performed on the various Cluster Nodes registered in the system.

CAL programs can be developed using the CAL Editor available through the BalticLSC FrontEnd component, as illustrated in Fig. 9. The editor is web-based and implements the full syntax of CAL. Individual Computation Modules can be added to the editor's toolbox, placed on the canvas and their pins connected through data flows. The editor assures dynamic validation of syntax, not allowing for incorrect connections. CAL diagrams are dynamically stored in the DiagramRegistry component through an appropriate API. Note that a detailed discussion of the FrontEnd component and

[1] www.balticlsc.eu.

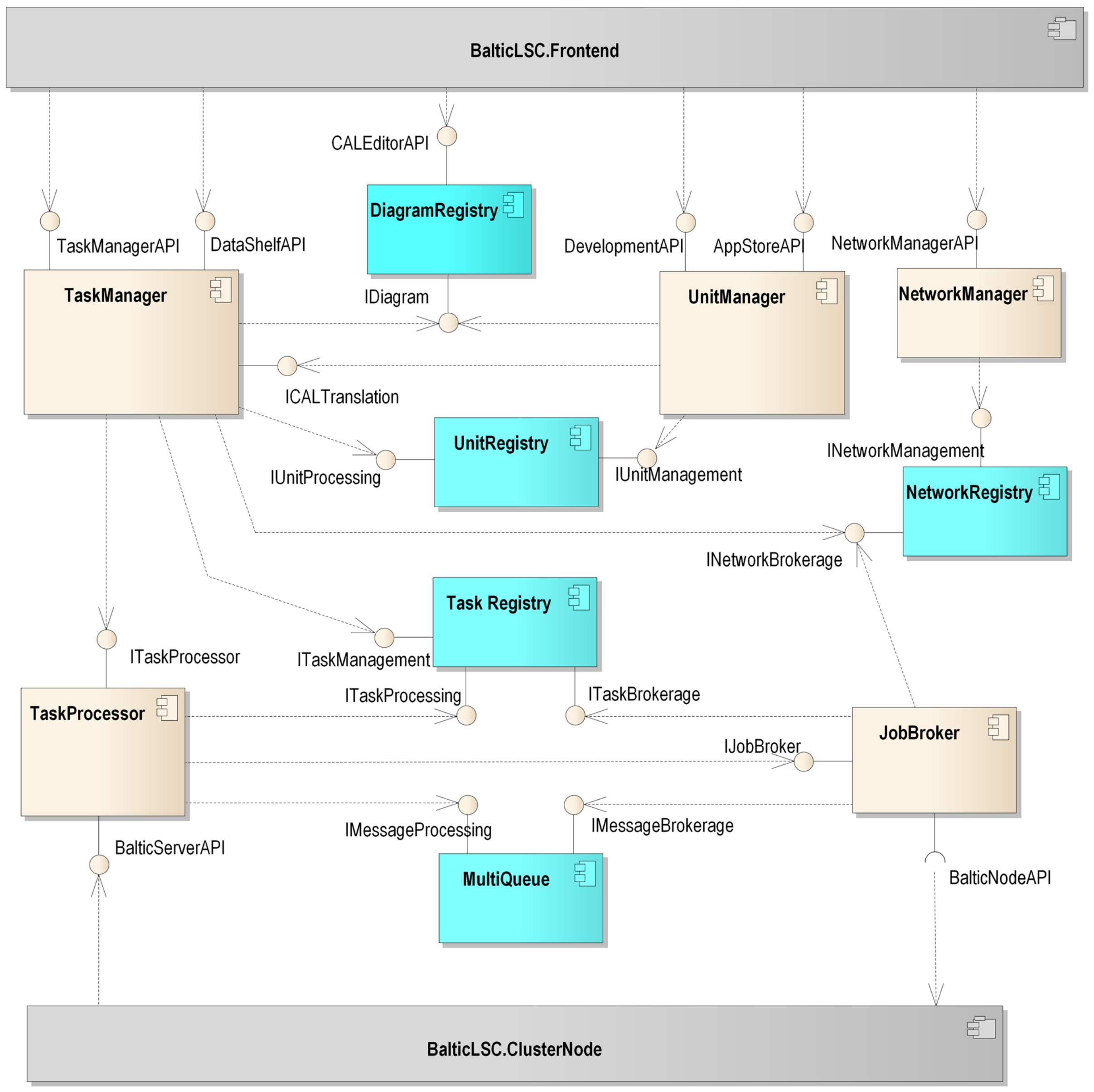
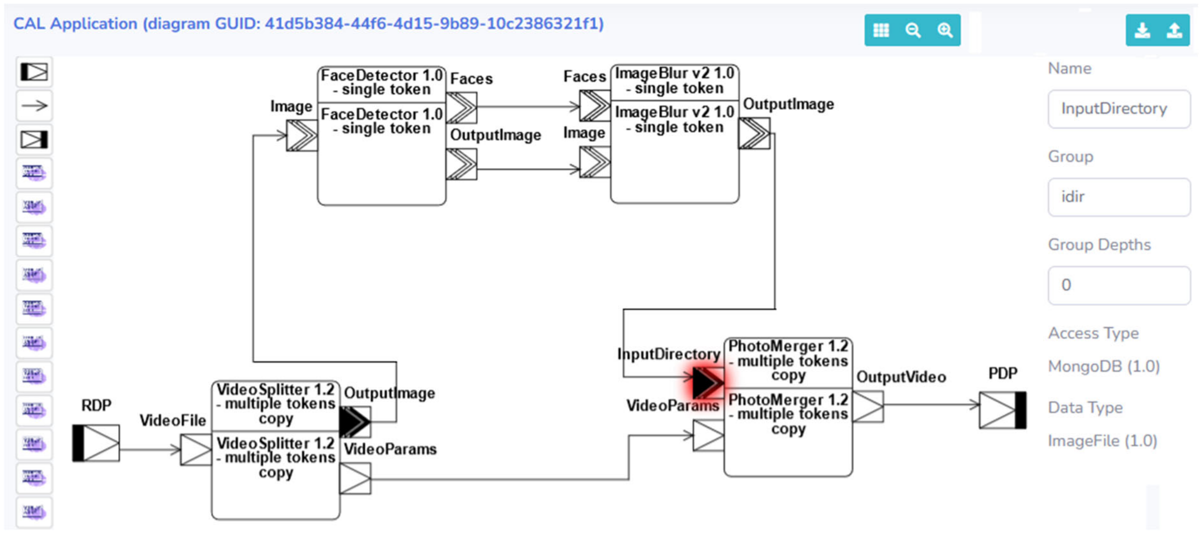


Fig. 8 Master node structure

the design details of the CAL Editor is out of the scope of this paper.

Applications expressed in CAL can be run from the BalticLSC Computation Cockpit illustrated in Fig. 10. The user can select an application and initiate a new task. All the current and finished tasks can be accessed and examined. For example, Fig. 11 shows one of the task executions (X8) from Fig. 10. As we can see, one of the job instances in this task has failed and the user can diagnose the problem by examining the final message and the appropriate logs (not shown in the figure).

Whenever a new task instance is created, appropriate CAL diagrams are accessed through the IDiagram interface of the DiagramRegistry component (see again Fig. 8). The CAL program associated with the given application is first processed by the TaskManager

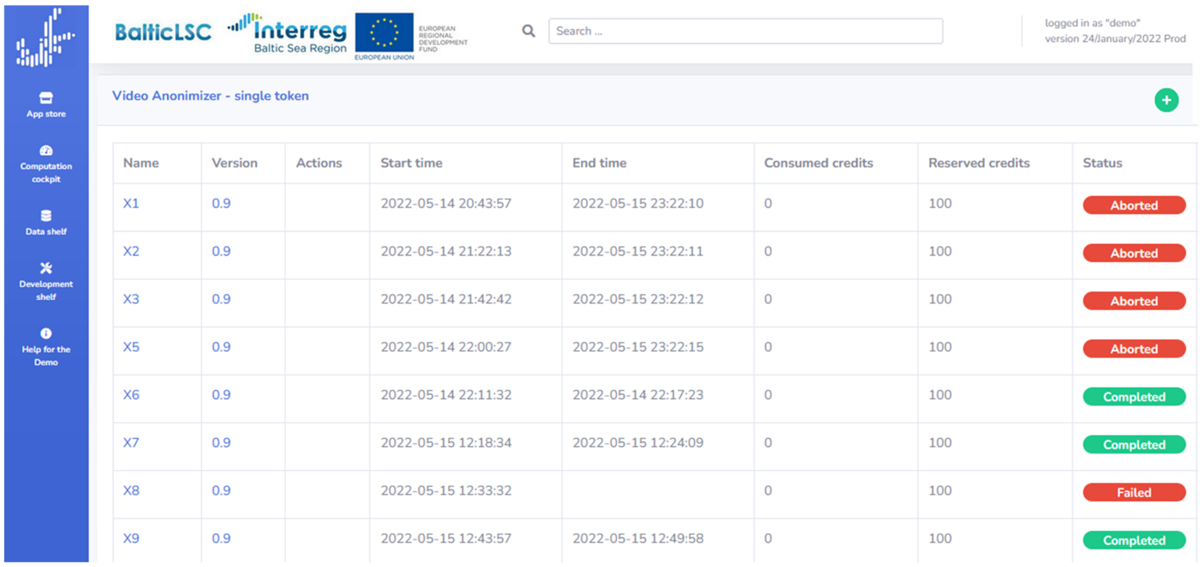

CAL Application (diagram GUID: 41d5b384-44f6-4d15-9b89-10c2386321f1)
FaceDetector 1.0 - single token
FaceDetector 1.0 - single token
Image
Faces
OutputImage
Faces
Image
ImageBlur v2 1.0 - single token
ImageBlur v2 1.0 - single token
OutputImage
InputDirectory
PhotoMerger 1.2 - multiple tokens copy
PhotoMerger 1.2 - multiple tokens copy
VideoParams
OutputVideo
PDP
RDP
VideoFile
VideoSplitter 1.2 - multiple tokens copy
VideoSplitter 1.2 - multiple tokens copy
OutputImage
VideoParams

Name: InputDirectory
Group: idir
Group Depths: 0
Access Type: MongoDB (1.0)
Data Type: ImageFile (1.0)

Fig. 9 CAL editor - example application

component. This component uses the translational semantics rules to translate from CAL to CAL-Exec and stores the result in the TaskRegistry component. It also uses the UnitRegistry component to access definitions of Computation Units.

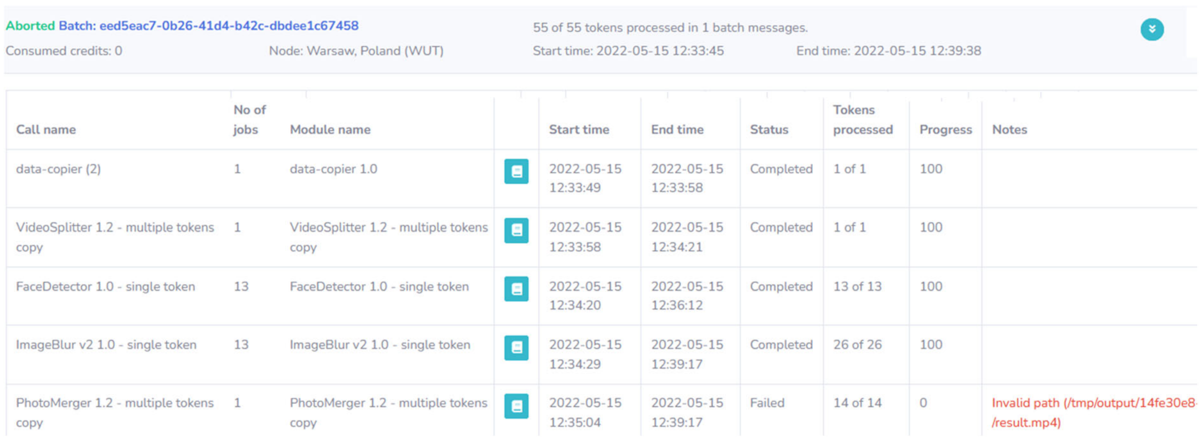
Following this, the TaskManager initiates the TaskProcessor component. This is done by passing DataToken instances received from the Frontend (specified by the user). The TaskProcessor accesses the CAL-Exec code stored in the TaskRegistry. Based on the received DataTokens, it interprets the CAL-Exec code to start JobBatch and Job instances. This is done with the help of the MultiQueue component. All the token instances are pushed to specific queues, which form groups that trigger respective job instances. This triggering is done according to the operational semantics of CAL-Exec. The queue component helps in managing multiple tokens that need to be directed to appro-



Video Anonimizer - single token

Name	Version	Actions	Start time	End time	Consumed credits	Reserved credits	Status
X1	0.9		2022-05-14 20:43:57	2022-05-15 23:22:10	0	100	Aborted
X2	0.9		2022-05-14 21:22:13	2022-05-15 23:22:11	0	100	Aborted
X3	0.9		2022-05-14 21:42:42	2022-05-15 23:22:12	0	100	Aborted
X5	0.9		2022-05-14 22:00:27	2022-05-15 23:22:15	0	100	Aborted
X6	0.9		2022-05-14 22:11:32	2022-05-14 22:17:23	0	100	Completed
X7	0.9		2022-05-15 12:18:34	2022-05-15 12:24:09	0	100	Completed
X8	0.9		2022-05-15 12:33:32		0	100	Failed
X9	0.9		2022-05-15 12:43:57	2022-05-15 12:49:58	0	100	Completed

Fig. 10 Computation Cockpit - example tasks for an application


Aborted Batch: eed5eac7-0b26-41d4-b42c-dbdee1c67458
55 of 55 tokens processed in 1 batch messages.
Consumed credits: 0
Node: Warsaw, Poland (WUT)
Start time: 2022-05-15 12:33:45
End time: 2022-05-15 12:39:38

Call name	No of jobs	Module name		Start time	End time	Status	Tokens processed	Progress	Notes
data-copier (2)	1	data-copier 1.0	▤	2022-05-15 12:33:49	2022-05-15 12:33:58	Completed	1 of 1	100	
VideoSplitter 1.2 - multiple tokens copy	1	VideoSplitter 1.2 - multiple tokens copy	▤	2022-05-15 12:33:58	2022-05-15 12:34:21	Completed	1 of 1	100	
FaceDetector 1.0 - single token	13	FaceDetector 1.0 - single token	▤	2022-05-15 12:34:20	2022-05-15 12:36:12	Completed	13 of 13	100	
ImageBlur v2 1.0 - single token	13	ImageBlur v2 1.0 - single token	▤	2022-05-15 12:34:29	2022-05-15 12:39:17	Completed	26 of 26	100	
PhotoMerger 1.2 - multiple tokens copy	1	PhotoMerger 1.2 - multiple tokens copy	▤	2022-05-15 12:35:04	2022-05-15 12:39:17	Failed	14 of 14	0	Invalid path (/tmp/output/14fe30e8 /result.mp4)

Fig. 11 Computation Cockpit - example task execution

priate job instances. For instance, some tokens with the same token number need to be transported to the same job instance, and others need to be distributed between several job instances. This mechanism is to some extent similar to that found in WS-PGRADE [45] but uses multi-level sequence identifiers contained in the tokens (metadata).

Token instance distribution done by the TaskProcessor in cooperation with the MultiQueue is illustrated in Figs. 12 and 13. The figures show example task executions for the CAL-Exec programs from Figs. 6 and 7. Token instances are denoted by circles with numbers corresponding to the token numbers as specified in the respective CAL-Exec programs. Figure 12a shows an initial step in task execution. An instance of token no. 1 has already been provided by the user and is waiting in the MultiQueue (denoted by "MQ"). At this moment, an instance of token no. 2 is sent from the front end and inserted into the queue (denoted by a solid arrow). This causes the initiation (denoted by a dashed arrow) of a new job batch execution "be101". This is consistent with the definition of the respective job batch ("b002" in Fig. 6) that requires the arrival of tokens no. 1 and 2.

In the next instance, the arrival of tokens 1 and 2 causes the initiation of job execution "je104" according to the definition of job "j004" (see Fig. 12b). Following this, "je004" can start its execution and consecutively produces token instances according to its job definition. Since the provided (output) DataTokens of "j004" are of "multiple token" type, the job execution can produce several tokens numbered "4" and "5". In our example, "je104" produces two sequences of two tokens which are shown in Fig. 12c. Tokens in a sequence are additionally numbered by the execution engine to keep track of token ordering (sequence numbering is denoted by numbers in squares; the final token in a sequence is denoted by an "f").

Figure 12d shows the status of task execution during the processing of token sequences produced by "je004". Based on the initiation rules for jobs "j005" and "j006", the execution engine starts several instances of these jobs ("je105", "je205", "je106" and "je206") and passes appropriate token instances to them. These new job executions start processing and finally provide tokens according to the definitions of jobs "j005" and "j006". As shown in Fig. 12e, the sequence numbers created by "je004" are maintained by the execution engine. After providing tokens on their output, job instances are terminated.

Figure 12f illustrates one of the further steps in token processing. It shows the initiation of an instance of the job "j007". It can be noted that the MultiQueue component takes care of grouping token instances according to sequence numbering. Thus, job execution "je107" is created only after the arrival of tokens numbered "6" and "7" with the same sequence numbers (here: "2f"). Consecutive groups of matching tokens initiate consecutive job executions, which is illustrated in Fig. 12g.

Figure 12h and i show the final steps in the example task execution. Job executions "je107" and "je207" produce tokens numbered "3", still maintaining the additional sequence numbering started by "je104". At this point, we should note that the output DataToken of the job "j007" is typed "single token" and "single

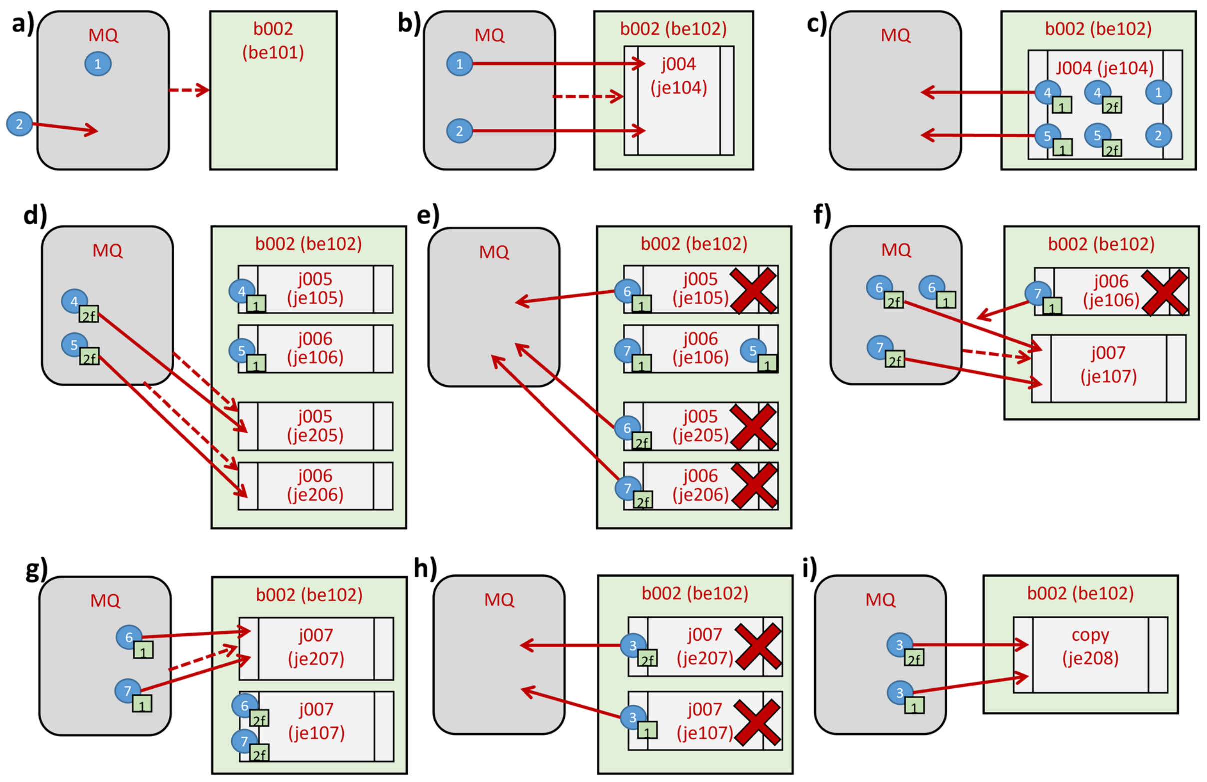


Fig. 12 Example flow of tokens in task execution ("strong" mode)

data" (token no. 3, see Fig. 6 again). At the same time, the output DataToken of batch "b002" (also no. 3) is typed "single token" and "multiple data". This means we need an additional job that will gather individual data items sent by the job executions of job "j007" and place them in a data folder. The additional job execution ("je208") is shown in Fig. 12i. It is equivalent to a job with an input DataToken typed as "multiple token" and "single data" and produces a single output token typed as "multiple data" (a folder).

Figure 12 is silent on the actual flow of data (files) which obviously follows the flow of tokens. The BalticLSC execution environment handles data transfer between external storage and the computation nodes on which job batches are executed. In our example, an appropriate copying job is executed when tokens no. 1 and 2 are delivered. It copies files specified by these tokens to the internal storage of the appropriate container holding the appropriate batch execution (here: "be102"). Since all the jobs in our first example are computed within one batch execution (within a single computation node), there is no need to copy any data. The tokens simply pass pointers to appropriate data elements kept in the internal storage. Finally, when a token is produced by the job execution "je208", an appropriate copying job sends the resulting folder (holding files specified by the tokens with no. 3) to an appropriate output storage. This way, the user can access the results of computations.

The issue of data transfer and job execution becomes more complex when a given task is executed in "weak" mode. This is illustrated in Fig. 13, which shows some key steps in the execution of a task based on the program from Fig. 7. Figure 13a shows the situation where "je108" has produced tokens no. 4 and 5, and appropriate jobs are being created. In this mode, it causes the creation of another batch execution which can be assigned to a different (possibly geographically distant) computation node. In Fig. 13b, we can notice the next step, in which another pair of tokens with a different sequence number ("2") is produced. This, in turn, causes yet another batch execution ("be204") to be created. Note that the execution environment makes sure to start the various executions of jobs "j009" and "j010"

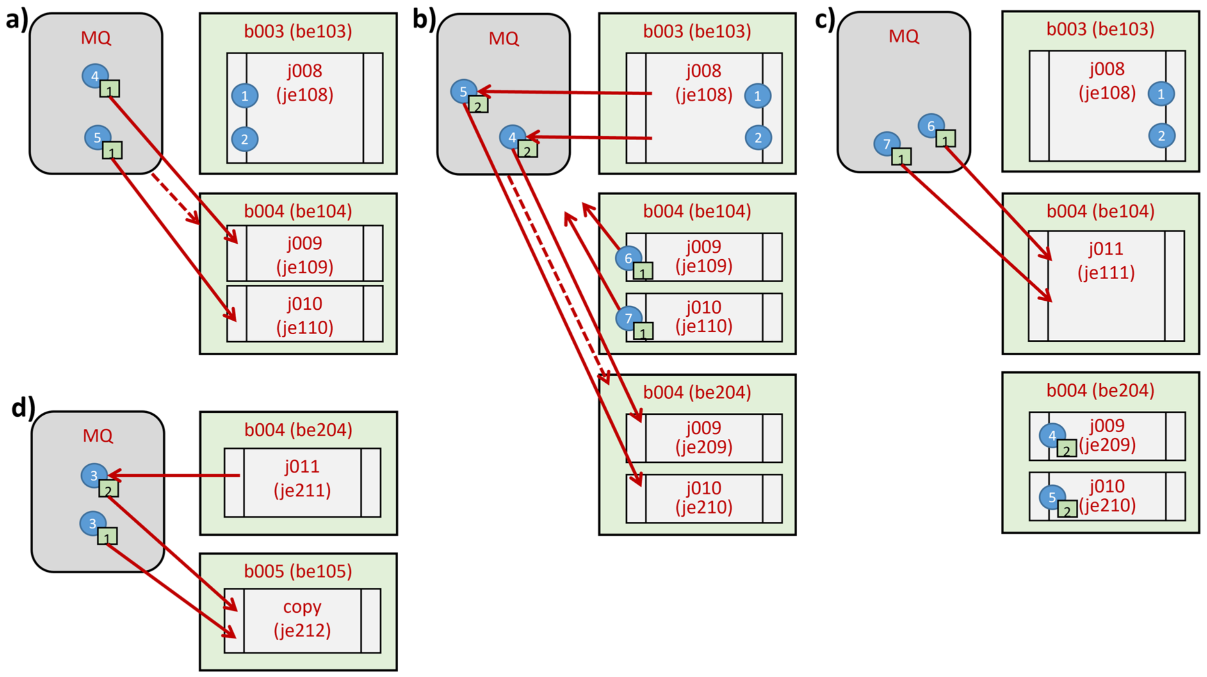


Fig. 13 Example flow of tokens in a task execution ("weak" mode)

together in the same batches, keeping track of their sequence numbering. This is consistent with the CAL-Executable program in Fig. 7, which assures that jobs "j009", "j010" and "j011" are kept together.

Figures 13c shows the situation where an execution of job "j011" is created in the same batch execution as the previously executed (and now terminated) executions of jobs "j009" and "j010". Finally, Fig. 13d shows the final step, where an additional copying job is created in a separate batch execution. Similarly to the previous example, all the tokens numbered "3" are directed to this new job. The distribution of batch executions potentially between several computation nodes necessitates additional data transfer. Thus, the execution environment introduces additional copying jobs. It keeps track of the various batch executions and assures that appropriate data elements (files, folders), are copied between the containers holding these distributed batch executions.

Individual batch executions and contained job executions are assigned to specific computation nodes (or "cluster nodes"). This is done through the JobBroker component shown in Fig. 8. This component has access to the NetworkRegistry that holds information about available cluster nodes. When a new batch execution is to be started, the JobBroker checks the resources available in each node and compares them with the resources required by the batch execution (determined from the contained jobs). Following this, it sends a batch-starting message to a selected cluster node. Note that the algorithm for assigning batches to cluster nodes is out of the scope of this paper.

Each cluster node is equipped with an installation of a container execution environment (currently, the system supports Kubernetes and Docker Swarm) and mechanisms for managing job batches assigned for execution on the given cluster - see Fig. 14. Communication between the master node and the cluster nodes is based on DataToken instances. These tokens are passed through appropriate interfaces - the BalticNodeAPI and the BalticServerAPI. In addition, the BalticNodeAPI allows for passing appropriate messages for starting and terminating Job Batch and Job instances. This is managed on each cluster by the BatchManager components. All the batch/job instance initiation and termination actions are scheduled by the ClusterManager components that interact directly with appropriate container execution environments (Kubernetes or Docker Swarm). Ultimately, these mechanisms allow for the parallel execution of containerised job instances. Each

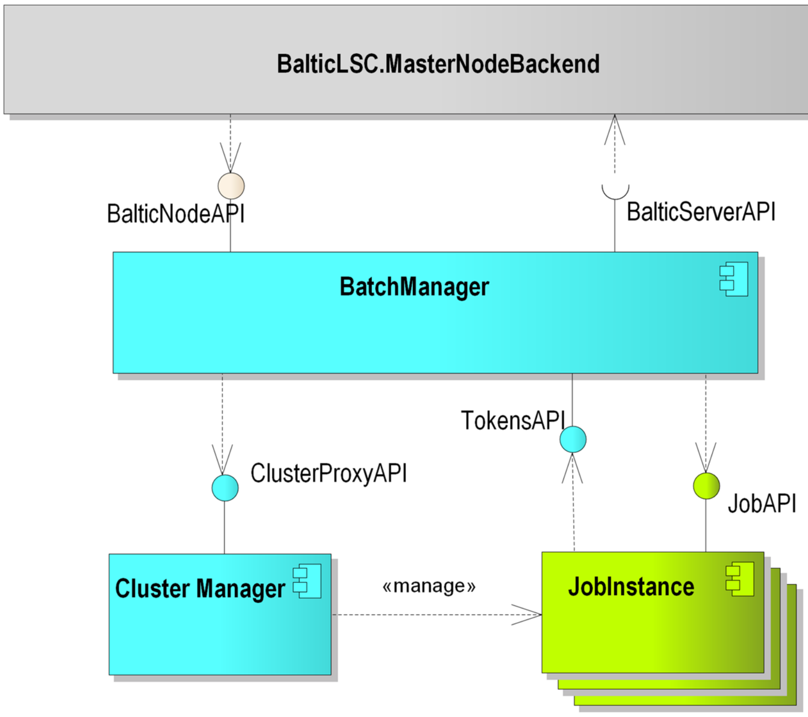


Fig. 14 Cluster node structure

JobInstance container implements a JobAPI that allows for the handling of DataToken instances by the individual job instance. The tokens produced by job instances are sent through the TokensAPI, implemented by the BatchManager.

JobInstance containers implement computation module releases (see Fig. 3). Each such module should be built as containerised software that receives the input data, performs the task, and sends the output data. However, there are no restrictions on what technologies (operating systems, programming languages, frameworks, etc.) are used. Communication with the BalticLSC Environment has to be done using the above-mentioned APIs. As was described, the communication between modules and the system is done using data tokens. Thus, a module should implement and use several predefined methods as REST API endpoints and read the appropriate configuration data from environment variables.

There are just two methods to be implemented by a module:

- PROCESSTOKENMESSAGE - accepts a message containing an input data token and responds with a simple integer denoting the initial status of token validation;
- GETSTATUS - responds with an appropriate job status object denoting the current status of data processing.

Moreover, there are just two methods to be used by a module:

- PUTTOKENMESSAGE - sends a message containing an output data token and receives a simple integer response;
- ACKTOKENMESSAGES - sends a special message acknowledging the completion of a complete computation execution..

In short, the code of the computation module should comply with the following life cycle.

1. Read appropriate configuration data and set up connections with the infrastructure (data stores, API endpoints, etc.)
2. Receive one or more input data tokens on the ProcessTokenMessage endpoint.
3. Perform the designated task - process input data and build output data tokens.
4. Send an output data token using the PutTokenMessage endpoint whenever one is ready.

5. When the computation execution ends, send an acknowledgement message using the AckTokenMessages endpoint.

To make the development of the modules easier, we provide templates for C# and Python, which hide all the technical details related to communication through the REST API and storing data in remote storage. More information about the development of computation modules can be found in the technical documentation in the "Download" section of the BalticLSC website.

7 Case Study Example - Waste Collection Logistics Optimisation

To show the applicability of CAL to solving various computation problems, we will present a real-life case study. The aim of the case study is to show how the BalticLSC Environment and CAL could be used to handle non-trivial large-scale computing tasks. The emphasis was on the CAL's ability to combine standalone computation modules into a single app and reuse the built apps and modules for other tasks.

The case study involves a company that develops a software component that optimises routes of waste collection vehicles. For the given set of customers, the vehicle fleet, and the waste fields, we need to compute an optimal (as short as possible) route. The vehicle capacity and the customer demand are considered while doing the optimisation. This results in solving the Capacitated Vehicle Routing Problem (CVRP) [61]. Since such computation problems are NP-hard, they take significant time and computation resources.

The task can be split into three main steps:

1) The coordinates (latitude and longitude) have to be found for the set of geographical objects (clients, vehicle depots, waste fields) given as addresses. This is called geocoding.
2) The distance matrix containing the road distances between all geographical objects in the task has to be computed.
3) Optimisation of the route has to be performed considering the capacity of vehicles, client demand and road distances between clients, waste fields, and depots.

The data model of modules' inputs and outputs is depicted in Fig. 15. There are two main classes (besides road maps) of objects that the modules are operating on. The first is the XGeoWasteLogisticsObject class. Instances of this class contain domain-specific (e.g., capacities of vehicles and amount of clients' demand) and geographical information (addresses). They are used as the input to the application we have built for the use case. The second is the XLocation class which describes the geographical location - longitude and latitude. The XDistance class instances refer to these objects but contain the actual road distance between these objects. Information on actual distances and coordinates is used internally by the computation application.

For each step, an independent computation module has been built according to the description in the previous section.

1) **Geo Coder**. The module requires a list of objects with addresses (XAddressable instances) and provides a list of coordinates (XLocationObject instances) for these objects. The module uses an external service – the OpenCage GeoCoding API (https://opencagedata.com/api).
2) **Geo Router**. The module finds the shortest distance between all given locations using the road network given by the map of the region where the locations are situated. Thus, the Geo Router module requires the list of locations (e.g., XLocationObject instances provided by the Geo Coder) and an OpenStreetMap file describing the region. The Geo Router provides a list of distances between objects (XDistance instances). This, in fact, forms the distance matrix for these objects. The Geo Router uses the open-source routing engine GraphHopper (https://www.graphhopper.com/).
3) **Geo Waste Logistics Optimizer**. The module optimises the route for the given set of vehicles, customers, and waste fields. Thus, the module requires a list of the mentioned objects (XGeoWasteLogisticsObject instances) and the distance matrix (distances between all the objects described by XDistance instances) for these objects. It provides a list of optimised routes - sequences of the customers that the vehicles should visit in the given order. The module uses the open-source optimisation engine OptaPlanner (https://www.optaplanner.org/).

Firstly, we build a computation application - the Distance Matrix Calculator (see Fig. 16) that computes the distance matrix for the given set of addressable objects

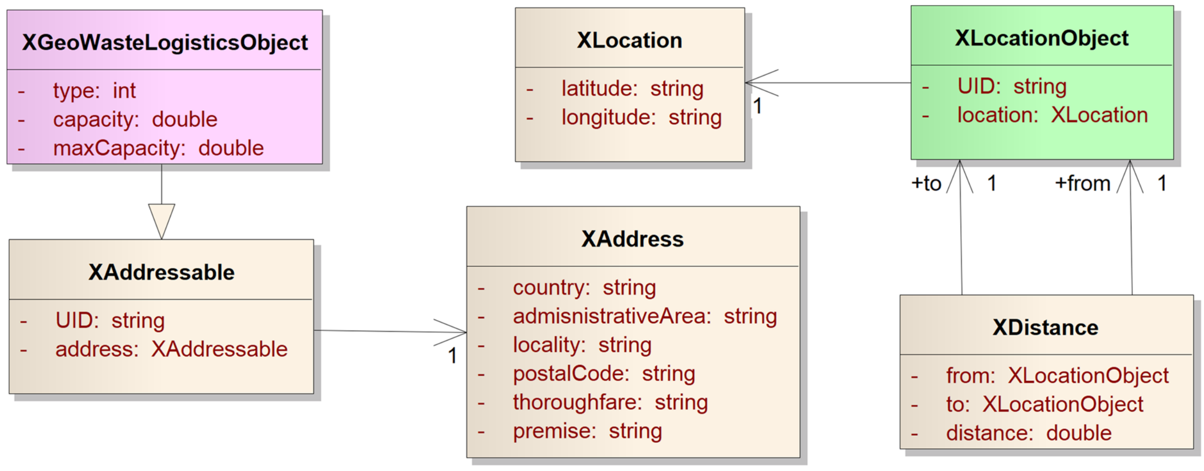
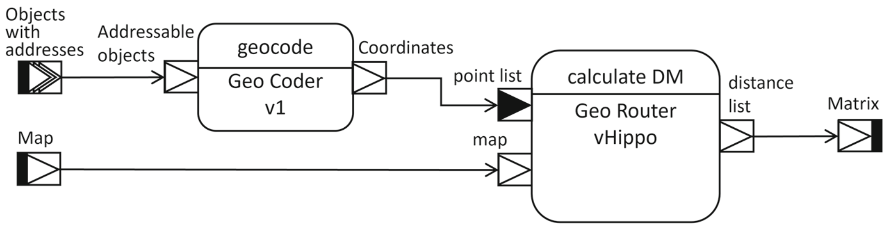


Fig. 15 Waste Collection Logistics Optimisation - data model

and the road map of the region in which these objects are located. This app can be used independently of this use case whenever a distance matrix is needed. There are two required (input) data pins. The “Objects with addresses” data pin is of “multiple data” type, while the “Map” data pin is “single data”. Thus, the app receives a single map file and multiple data files containing the objects’ information. The usage of the “multiple data” pin allows the BalticLSC System to split data and parallelize the execution of the Geo Coder module to multiple job instances. Unlike in the previous examples, where the splitting and merging of computation modules provided the possibility of concurrent execution of job instances, the splitting is hidden behind the mismatch of data multiplicity of data pins on the opposite sides of the data flow. Since the Geo Coder module processes just one data item (token) at a time, the execution of the module (regardless of whether concurrent instances are present or not) produces a sequence of new data items (tokens) on the provided (output) data pin of the module. Thus, the next module, Geo Router, has a required “multiple token, single data” pin called “point_ list”. It collects all the produced coordinates and, in fact, acts as a merger. The Geo Router also requires a map passed straight from the declared data pin. The module produces a list of distances – the distance matrix passed to the app’s provided declared data pin called “Matrix”.

Secondly, we reuse the distance matrix calculation app and build the Waste Collection Logistics Optimisation Application (see Fig. 17). The app has three required data pins (inputs). Two are needed to pass the data to the Distance Matrix Calculator (“Objects with addresses” and “Map”). The third pin is used to pass domain objects to the Geo Waste Logistics Optimizer module. In fact, it would be enough to have just one data pin instead of the “Objects with capacities” and “Objects with addresses” (they are copies of the same objects). However, due to the limitations of the data transformation capabilities of CAL, two pins are required. Note that the data multiplicities of both pins differ. The Geo Waste Optimizer module has two required data pins. The “AllGeoObjects” data pin receives a single file containing all the domain objects, while the “DistanceMatrix” data pin receives a single file containing the distance list between these objects

Fig. 16 Distance Matrix Calculator Application



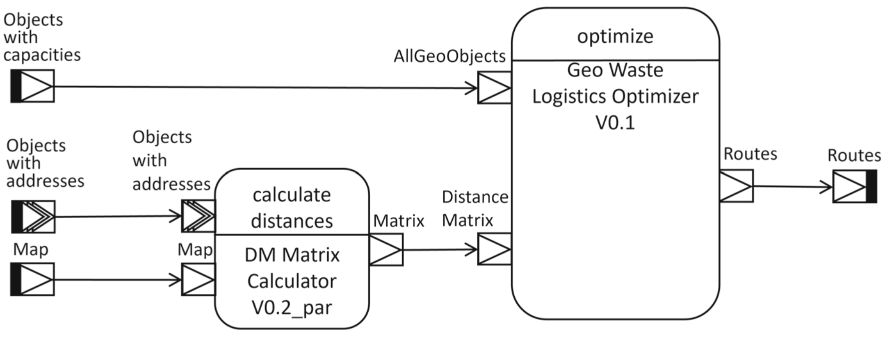


Fig. 17 Waste Collection Logistics Optimisation Application

produced by the Distance Matrix Calculator app. The result is a single file containing sequences of domain objects as routes for each vehicle in the task. It is passed through the provided declared data pin "Routes,".

The application and its individual modules have been implemented and tested and are currently available for reuse in the BalticLSC system. Through this case study, we have shown how computation modules can be reused for multiple computation applications and how a computation application can be reused in another computation application. The usage of the built modules is much broader than just the waste collection optimisation domain. Distance matrices, and generally – distances between geographical locations, are needed in a wide range of domains related to transportation and logistics. Even each of the modules separately is reusable. Geo-coding, as well as geo-routing, can be useful for different purposes, e.g., for GIS analysis and cartography. Thus, thoughtfully chosen "bricks" (in the case of BalticLSC the developed computation modules) can serve as building blocks for a wide range of possible computation applications in various domains. CAL and its execution environment is a usable "glue" to make them work together in parallel without the need for a steep learning curve or deep knowledge of underlying technical details.

8 Conclusion and Future Work

The presented general-purpose language allows the definition of distributed and parallel computations in a visual, low-code way. It has simple graphical syntax and precise runtime semantics. The language implementation comprises an online graphical editor and a comprehensive execution environment (BalticLSC). An important characteristic of CAL is that its programs operate at a high level of abstraction, where the fundamental entities are reusable computation units. Synchronisation of computations is based on flowing data that arrive at the inputs of specific unit instances. The flow of computations in a CAL application is controlled by data produced by the computation units. This characteristic of CAL allows for the automatic distribution of computations and optimisation of data transfer between computation nodes.

Developing a CAL-based Large Scale Computing application requires programming at two distinct levels. The first level uses a visual language that people can use without advanced knowledge of distributed programming and parallelization. At this level, the programmer concentrates on the actual computation algorithm in terms of high-level computation steps and data flowing between these steps. The second level is the development of computation modules. This requires typical programming and technical skills but does not require advanced parallel and distributed programming. The developed modules can be reused easily within CAL programs, thus avoiding code duplication.

From the perspective of the CAL user, the programming task consists in selecting and reusing "computation blocks", and then defining data flows between these blocks. CAL programmers can reflect the flow of data between different computation steps in a natural way. At the same time, the execution environment allows for easy management and structuring of user's data sets which reflect specific data types. Additionally, the "computation block" structure facilitates the reusability of code. The CAL programmer can reuse computation modules, use entire computation applications as computation modules in new applications, and even reuse data sets between different computation tasks. On the other hand, module developers can easily reuse existing software, e.g. in the form of an existing library, by incorporating them directly into a computation module. This makes the solution available to every CAL programmer in the future.

Thanks to its data-flow orientation and the online execution environment, CAL abstracts away all technicalities associated with parallelization and distribution of computing (even across many computation clusters – cf. batches). Additionally, the data-flow orientation of the language allows for easy "serial" parallelization of computations by automatically processing multiple tokens simultaneously without the prerequisite of parallelization knowledge from the end-user. Therefore, the end-user can focus on the complexity of data, its dependencies and processing without manually managing computation parallelization and orchestration.

It can be noted that the individual computation modules (processing steps) can be designed with varying levels of granularity. However, one must remember that a computation module is implemented as a container. Thus, this granularity should be a manageable size. This also influences the granularity of parallel processing. It is controlled mainly by the data flowing between computation module instances (jobs). Thus, the proper design of data pins (varying token and data multiplicities) is crucial for allowing the BalticLSC Environment to decide the number of parallel workers to be launched for the same job depending on the availability of computation resources.

The granularity of processing is also related to the performance of computations. We have not done specific performance tests for the CAL implementation. This is because the overall performance is determined by the performance of computation nodes, the efficiency of the computation modules code, containerisation environments, and data transfer to/from the nodes. Execution logs collected in case studies show that the times used for diagram translation and job brokerage are minimal compared to the processing times and have little or no impact on the total execution time. Thus, the graphical nature of CAL does not significantly influence the performance of computations.

Another important related issue is the performance of application development. The built-in reusability of code in the form of computation modules and automatic computation orchestration has a significant potential to reduce the work required to develop computation software, similar to other low-code solutions. The speed-up should be much higher when many required computation modules are already available on the BalticLSC platform. To foster this goal, CAL can be enhanced by adding automatic transformation of data between modules, allowing for greater flexibility and reusability of existing computation modules.

The future research agenda will be mainly based on the development of further computation modules and improving the usability of CAL. This would allow validating reduction of effort when using the CAL environment with a significant portfolio of reusable computation modules. To conduct such validation, the research agenda would include experimental work comprising controlled experiments comparing developers' performance using CAL and traditional programming models. Current results show promising results but are based on anecdotal evidence. CAL has already been used in several industrial applications and several student projects (including Master's degree projects). All these examples show the usability of the language and the relative ease of developing computation modules based on existing computation libraries. However, this needs to be systematically analysed, which will be the subject of future research and publications.

Another interesting issue related to CAL that is worth researching is the enhancement of CAL's flexibility regarding the execution environment. This would consist in shifting from language interpretation (as in BalticLSC) to its compilation. This would allow the generation of "autonomous workflows" according to the choreography approach, similar to that proposed by the Flowbster system [47]. Compilation of CAL programs into other workflow specification formats would also allow for integration with other distributed computation ecosystems (like Galaxy, WS-PGRADE or Flowbster), similar to that proposed by GeWWE [48].

In summary, in the future, we would like to enhance and validate CAL's potential to significantly foster the use of Large Scale Computations. This is especially important in the current world where programming and especially advanced specialised programming skills are scarce on the market. Based on our current results, we can claim that CAL limits the required knowledge and experience needed to develop di stributed applications. Instead, it allows the language end-users to focus on the complexity of the data they are working with, easing the parallelization and orchestration process. In effect, we demonstrate how the low-code approach can be used to define end execute workflows in distributed computing.

Author contributions Kamil Rybiński and Michał Śmiałek are the primary authors of the language, have contributed to its syntax and defined its semantics. Agris Sostaks have contributed to the language syntax and semantics and have prepared the syntax descriptions and the case study example. Krzysztof Marek has contributed to language usability and has participated in formulating textual content. Radosław Roszczyk and Marek Wdowiak have contributed to language implementation and have participated in formulating textual content. All authors read and approved the final manuscript

Funding This work is partially funded by the European Regional Development Fund, Interreg Baltic Sea Region Programme, project BalticLSC #R075

Data Availability Not applicable

Declarations

Ethics approval and consent to participate Not applicable

Consent for publication Not applicable

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article

References

1. Richardson, C., Rymer, J.R., Mines, C., Cullen, A., Whittaker, D.: New development platforms emerge for customer-facing applications. Forrester report (2014)
2. Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Software and Systems Modeling **21**(2), 437–446 (2022). https://doi.org/10.1007/s10270-021-00970-2
3. Trigo, A., Varajão, J., Almeida, M.: Low-code versus code-based software development: Which wins the productivity game? IT Professional **24**(5), 61–68 (2022). https://doi.org/10.1109/MITP.2022.3189880
4. Noone, M., Mooney, A.: Visual and textual programming languages: a systematic review of the literature. Journal of Computers in Education **5**(2), 149–174 (2018). https://doi.org/10.1007/s40692-018-0101-5
5. Kuhail, M.A., Farooq, S., Hammad, R., Bahja, M.: Characterizing visual programming approaches for end-user developers: A systematic review. IEEE Access **9**, 14181–14202 (2021). https://doi.org/10.1109/ACCESS.2021.3051043
6. Silva, M., Dias, J..P., Restivo, A., Ferreira, H..S.: A review on visual programming for distributed computation in IoT. In: Paszynski, M., Kranzlmüller, D., Krzhizhanovskaya, V..V., Dongarra, J..J., Sloot, P..M..A. (eds.) Computational Science – ICCS 2021, pp. 443–457. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77970-2_34
7. Cabot, J., Clariso, R.: Low code for smart software development. IEEE Software **40**(1), 89–93 (2023). https://doi.org/10.1109/ms.2022.3211352
8. Peltz, C.: Web services orchestration and choreography. Computer **36**(10), 46–52 (2003). https://doi.org/10.1109/mc.2003.1236471
9. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
10. Marek, K., Śmiałek, M., Rybiński, K., Roszczyk, R., Wdowiak, M.: BalticLSC: Low-code software development platform for large scale computations. Computing and Informatics **40**(4), 734–753 (2021). https://doi.org/10.31577/cai_2021_4_734
11. Roszczyk, R., Wdowiak, M., Smialek, M., Rybinski, K., Marek, K.: BalticLSC: A low-code HPC platform for small and medium research teams. In: 2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2021). https://doi.org/10.1109/vl/hcc51201.2021.9576305
12. Moody, D.: The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering **35**(6), 756–779 (2009). https://doi.org/10.1109/tse.2009.67
13. Amaral, V., Norberto, B., Goulão, M., Aldinucci, M., Benkner, S., Bracciali, A., Carreira, P., Celms, E., Correia, L., Grelck, C., Karatza, H., Kessler, C., Kilpatrick, P., Martiniano, H., Mavridis, I., Pllana, S., Respício, A., Simão, J., Veiga, L., Visa, A.: Programming languages for data-intensive HPC applications: A systematic mapping study. Parallel Computing **91**, 102584 (2020). https://doi.org/10.1016/j.parco.2019.102584
14. Diaz, J., Munoz-Caro, C., Nino, A.: A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on Parallel and Distributed Systems **23**(8), 1369–1386 (2012). https://doi.org/10.1109/tpds.2011.308
15. Brooks, F.P.: No silver bullet essence and accidents of software engineering. Computer **20**(4), 10–19 (1987). https://doi.org/10.1109/mc.1987.1663532
16. Whitley, K.N.: Visual programming languages and the empirical evidence for and against. Journal of Visual Languages & Computing **8**(1), 109–142 (1997). https://doi.org/10.1006/jvlc.1996.0030
17. Cunniff, N., Taylor, R.P.: Graphical vs. textual representation: An empirical study of novices' program comprehen-

sion. In: Empirical Studies of Programmers: Second Workshop, pp. 114–131 (1987)
18. Navarro-Prieto, R., Cañas, J.J.: Are visual programming languages better? The role of imagery in program comprehension. International Journal of Human-Computer Studies **54**(6), 799–829 (2001). https://doi.org/10.1006/ijhc.2000.0465
19. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cognitive science **11**(1), 65–100 (1987)
20. Zernik, D., Snir, M., Malki, D.: Using visualization tools to understand concurrency. IEEE Software **9**(3), 87–92 (1992). https://doi.org/10.1109/52.136185
21. Zhang, K., Ma, W.: Graphical assistance in parallel program development. In: Proceedings of 1994 IEEE Symposium on Visual Languages, pp. 168–170 (1994). https://doi.org/10.1109/vl.1994.363628
22. Zhang, K., Hintz, T., Ma, X.: The role of graphics in parallel program development. Journal of Visual Languages & Computing **10**(3), 215–243 (1999). https://doi.org/10.1006/jvlc.1998.0109
23. Frost, R.: High-performance visual programming environments. ACM SIGGRAPH Computer Graphics **29**(2), 45–48 (1995). https://doi.org/10.1145/204362.204373
24. Lee, P., Webber, J.: Taxonomy for visual parallel programming languages. Technical report, University of Newcastle upon Tyne, Computing Science (2003)
25. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review **41**(3), 59–72 (2007). https://doi.org/10.1145/1272998.1273005
26. Li, G.-D., Zhang, D.F.: Dependency graphs embedding confluent graph grammars. Journal of Software **15**(7), 956–968 (2004)
27. Cao, J., Chan, A.T.S., Sun, Y.: GOP: A graph-oriented programming model for parallel and distributed systems. In: New Horizons of Parallel and Distributed Computing, pp. 21–36. Kluwer Academic Publishers, Boston (2005). https://doi.org/10.1007/0-387-28967-4_2
28. Browne, J.C., Azam, M., Sobek, S.: CODE: a unified approach to parallel programming. IEEE Software **6**(4), 10–18 (1989). https://doi.org/10.1109/52.31648
29. Newton, P., Browne, J.C.: The CODE 2.0 graphical parallel programming language. In: Proceedings of the 6th International Conference on Supercomputing- ICS '92 (1992). https://doi.org/10.1145/143369.143405
30. Browne, J.C., Hyder, S.I., Dongarra, J., Moore, K., Newton, P.: Visual programming and debugging for parallel computing. IEEE Parallel & Distributed Technology: Systems & Applications **3**(1), 75–83 (1995). https://doi.org/10.1109/88.384586
31. Stankovic, N., Kang, Z.: A distributed parallel programming framework. IEEE Transactions on Software Engineering **28**(5), 478–493 (2002). https://doi.org/10.1109/tse.2002.1000451
32. Galicia, J.C., Garcia, F.R.M.: Graphical specification language for distributed systems. In: 2006 15th International Conference on Computing (2006). https://doi.org/10.1109/cic.2006.39
33. McCormick, P., Inman, J., Ahrens, J., Mohd-Yusof, J., Roth, G., Cummins, S.: Scout: a data-parallel programming language for graphics processors. Parallel Computing **33**(10–11), 648–662 (2007). https://doi.org/10.1016/j.parco.2007.09.001
34. Böhm, S., Běhálek, M., Meca, O., Šurkovskỳ, M.: Visual programming of MPI applications: Debugging, performance analysis, and performance prediction. Computer Science and Information Systems **11**(4), 1315–1336 (2014). https://doi.org/10.2298/csis131204052b
35. Zhang, D.Q., Zhang, K.: A visual programming environment for distributed systems. In: Proceedings of Symposium on Visual Languages, pp. 310–317 (1995). https://doi.org/10.1109/vl.1995.520824
36. Kacsuk, P., Cunha, J., Dózsa, G., Lourenço, J., Fadgyas, T., Antao, T.: A graphical development and debugging environment for parallel programs. Parallel Computing **22**(13), 1747–1770 (1997). https://doi.org/10.1016/S0167-8191(96)00075-0
37. Delaitre, T., Zemerly, M.J., Justo, G., Audo, O., Winter, S.C.: EDPEPPS: an integrated environment for the parallel development life-cycle. Future Generation Computer Systems **16**(6), 585–595 (2000). https://doi.org/10.1016/S0167-739X(99)00072-2
38. Quiroz-Fabián, J.L., Román-Alonso, G., Castro-García, M.A., Buenabad-Chávez, J., Boukerche, A., Aguilar-Cornejo, M.: VPPE: A novel visual parallel programming environment. International Journal of Parallel Programming **47**(5), 1117–1151 (2019). https://doi.org/10.1007/s10766-019-00639-w
39. Meca, O., Böhms, S., Behálek, M., Jančar, P.: An approach to verification of MPI applications defined in a high-level model. In: 2016 16th International Conference on Application of Concurrency to System Design (ACSD), pp. 55–64(2016). https://doi.org/10.1109/ACSD.2016.17
40. Feng, A., Gardner, M., Feng, W.-c.: Parallel programming with pictures is a Snap. Journal of Parallel and Distributed Computing **105**, 150–162 (2017). https://doi.org/10.1016/j.jpdc.2017.01.018
41. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The Scratch programming language and environment. ACM Transactions on Computing Education (TOCE) **10**(4), 1–15 (2010). https://doi.org/10.1145/1868358.1868363
42. Böhm, S., Běhálek, M.: sage of Petri nets for high performance computing. In: Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing, pp. 37–48 (2012). https://doi.org/10.1145/2364474.2364481
43. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/b95112
44. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. Journal of Grid Computing **13**, 457–493 (2015). https://doi.org/10.1007/s10723-015-9329-8
45. Kacsuk, P., Farkas, Z., Kozlovszky, M., Hermann, G., Balasko, A., Karoczkai, K., Marton, I.: WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities. Journal of Grid Computing **10**(4), 601–630 (2012). https://doi.org/10.1007/s10723-012-9240-5

46. Kacsuk, P.(ed.):Science Gateways for Distributed Computing Infrastructures. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11268-8
47. Kacsuk, P., Kovács, J., Farkas, Z.: The flowbster cloud-oriented workflow system to process large scientific data sets. Journal of Grid Computing **16**(1), 55–83 (2018). https://doi.org/10.1007/s10723-017-9420-4
48. Gesing, S., Atkinson, M., Klampanos, I., Galea, M., Berthold, M.R., Barbera, R., Scardaci, D., Terstyanszky, G., Kiss, T., Kacsuk, P.: The demand for consistent web-based workflow editors. In: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, pp. 112–123. Association for Computing Machinery, Denver, Colorado (2013). https://doi.org/10.1145/2534248.2534260
49. Hariri, S., Kim, D., Kim, Y., Ra, I.:Virtual distributed computing environment. Technical Report AFRL-IF-RS-TR-2000-24, Syracuse University (2000)
50. Sampedro, Z., Hauser, T., Sood, S.: Sandstone HPC: A domain-general gateway for new HPC users. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact, pp. 1–7 (2017). https://doi.org/10.1145/3093338.3093360
51. Calegari, P., Levrier, M., Balczyński, P.: Web portals for high-performance computing: a survey. ACM Transactions on the Web (TWEB) **13**(1), 1–36 (2019). https://doi.org/10.1145/3197385
52. Tekinerdogan, B., Arkin, E.: ParDSL: a domain-specific language framework for supporting deployment of parallel algorithms. Software & Systems Modeling **18**(5), 2907–2935 (2019). https://doi.org/10.1007/s10270-018-00705-w
53. Bisong, E.: Kubeflow and Kubeflow pipelines. In: Building Machine Learning and Deep Learning Models on Google Cloud Platform, pp. 671–685. Apress, Berkeley (2019). https://doi.org/10.1007/978-1-4842-4470-8_46
54. Burns, B., Beda, J., Hightower, K.: Kubernetes: up and Running: Dive Into the Future of Infrastructure. O'Reilly Media, (2019)
55. Zandifar, M., Abdul Jabbar, M., Majidi, A., Keyes, D., Amato, N.M., Rauchwerger, L.: Composing algorithmic skeletons to express high-performance scientific applications. In: Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 415–424 (2015). https://doi.org/10.1145/2751205.2751241
56. Anderson, D.P.: BOINC: a platform for volunteer computing. Journal of Grid Computing **18**, 99–122 (2020). https://doi.org/10.1007/s10723-019-09497-9
57. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Setihome: An experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002). https://doi.org/10.1145/581571.581573
58. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, 1st edn. Addison-Wesley Professional, (2008)
59. Slonneger, K., Kurtz, B.L.: Formal Syntax and Semantics of Programming Languages. Addison-Wesley, (1995)
60. Plotkin, G.D.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming **60–61**, 17–139 (2004). https://doi.org/10.1016/j.jlap.2004.05.001
61. Toth, P., Vigo, D.: Models, relaxations and exact approaches for the capacitated vehicle routing problem. Discrete Applied Mathematics **123**(1–3), 487–512 (2002). https://doi.org/10.1016/S0166-218X(01)00351-1