Combining Flows and Rules in a Low-Code Platform for Smart Water Management

Jens Nicolay*, Bjarno Oeyen*, Samuel Ngugi Ndung'u*, Thierry Renaux*,
Maxime Démarest†, Boud Verbeiren†, Wolfgang De Meuter*
*Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
e-mail: {jens.nicolay | bjarno.oeyen | samuel.ngugi | thierry.renaux | wolfgang.de.meuter}@vub.be
†Hydria, Brussels, Belgium
e-mail: {mdemarest | bverbeiren}@bmwb.be

***Abstract*—The paper describes a low-code programming model and environment for automating sensor data processing pipelines for the smart water management domain. We identify visual flow-based programming and rule-based approaches as two promising avenues for building low-code programming models in this domain, but likewise identified a total of five problems faced by these approaches when applied to the domain. We propose a solution that tackles those problems, both as a high-level vision (combining the visual flow-based programming approach with rule-based approach, where each approach is applied for the programming tasks they are best suited for) and as a concrete design of a low-code programming model. We sketch our implementation, and discuss its limitations.**

Keywords-flow-based programming; rule-based programming; visual programming; low-code; sensors; internet of things.

I. Introduction

Low-code programming enables process automation by users that would otherwise not be able to automate their processes. In this paper, we describe a low-code programming model and environment for automating sensor data processing pipelines for the water management domain. Automating the decision making process can save time, but also reduces mistakes in tedious manual tasks [1]. Our goal is to provide the conceptual model and practical tools to enable domain experts (that do not need to be computer programmers) to define, reason about, and maintain software solutions that help them make accurate, timely decisions for managing surface water, sewer, and rainfall drainage infrastructure.

It is generally accepted that the *flow-based* programming paradigm [2] lends itself very well to the task of expressing data processing pipelines, and simultaneously lends itself well to visual programming. Many popular tools exist for visually developing and deploying data processing applications (e.g., Node-RED [3], NoFlo [4], etc.). Despite its advantages, we argue that it falls short at expressing common aspects of those pipelines. Non-trivial logic (e.g., aggregation and correlation) are better expressed as logical deductions in declarative *rule-based* programming paradigm. We found that a straightforward translation of traditional logic rules to a visual flow-based platform does not offer a satisfying solution in our water management scenarios.

The crux of our argument is as follows: if the goal of a visual low-code programming model is to be accessible by domain experts who are non-expert programmers, then the model should enable these domain experts to express the necessary logic with the least amount of friction. In this paper, we show that no single paradigm excels at capturing the concerns that our water domain experts wanted to express. We argue that domain experts need both a flexible system for transforming and filtering data, *and* a capable system for correlating and accumulating different measurements over time. We show that both paradigms can be unified into one uniform model that lends itself to a coherent two-layered low-code data processing platform. In our vision, the parts of a processing pipeline that lend themselves well to being expressed as a pipeline are expressed using a flow-based approach, while the parts that are best expressed as logical deduction are expressed using a block-based low-code programming layer backed by a rule engine.

The remainder of this paper is structured as follows. In Section II, we list two motivating scenarios, which we use to derive a problem statement in Section III. In Section IV, we describe our approach by stating how a visual programming environment combining flows and rules will tackles these problems. In Section V, we provide a bird's eye perspective on the platform, before presenting the conclusion and future work in Section VI.

II. Motivating Scenarios

Our motivation for the design of a uniform low-code programming platform is centred around two driving scenarios from the water management domain.

UC1 *The pre-validation of rainfall measurements.* Raw measurements from the sensor devices are analysed and corrected for known anomalies. Sensors need to be manually calibrated over time. While being calibrated, sensors produce faulty data (e.g., measuring rain when there should not be rain). This use case is concerned with identifying these calibration events, removing the raw measurements, and replacing the missing data with data from the statistically most-correlated, nearby, measurement station(s) that did have actual data.

UC2 *The real-time monitoring of surface water quality.* Specific locations and specific parameters (e.g., temperature, pH, conductivity, etc.) have their own safe ranges and expected behaviours. Both abnormal values (i.e., values that are not in a well-known safe range) and spikes (i.e., measurements that are significantly higher/lower than the previous one) need to be detected. An alert can then be sent out, by the

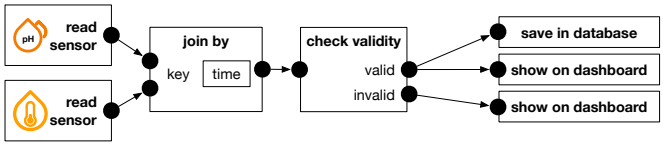


Figure 1: Example of a flow where traditional flow components shine: the shape of the flow intuitively conveys the outcome, the components' behaviours are straightforward to infer.

platform, to investigate the cause of the abnormal readings.

Before the introduction of our low-code platform, necessity dictated that the former scenario was tackled using spreadsheets, commonly recognised as one of the most widely used tools for low-code programming [5]. Unfortunately, this manual process was tedious and error-prone, and was never performed on live data. The latter was only possible in a limited way, and only on raw data as measured and stored into the sensors' data collection platform. I.e., not using any of the validated data as generated by the first use case.

A. Flow-Based Model

Some of the aspects of these use cases fit well in a traditional flow-based programming approach. For example, Figure 1 provides a sketch of a low-code program to validate acidity and temperature readings. By joining the data produced by two sensors, each time step can be validated individually. The general semantics of the program are easy-to-understand from this visualisation: flows consist of components that contain computations, and arrows between them connect the input and outputs of these computations. The flow-based model is, traditionally, well-suited for the development of distributed event-driven systems, such as data processing pipelines, IoT applications, and Cyber-Physical Systems.

B. Rule-Based Model

However, some implementation aspects of the presented use cases are better expressed using a rule-based programming approach. For example, for detecting spikes or applying inter-station correlation the rule-based approach is more favourable.

Rule-based logic or symbolic AI is concerned with expressing and representing human knowledge and logic in a declarative manner, usually based on facts and by specifying "if-then" rules that connect and manipulate attributes of those facts to produce new facts. Foregoing the low-code requirement for a moment, Figure 2 shows a rule-based approach for handling calibrations in UC1. The figure depicts a code-based approach in a variant of Datalog extended with stratified negation [6]. Rules provide fine-grained control over the generation of new facts from existing facts. For example, the `suspiciousRainfall` rule in Figure 2 denotes exactly when a rainfall measurement (of a given quantity `MM`, at a given time `T`, for a specific measurement station `S_ID`) is deemed suspicious: if there is at least 2 mm of rain but no rain at any other measurement station known by the system (i.e. within the same city). In general, rule-based logic is

```
1  rainfallAtStationOtherThan(T, S_ID) :=
2    rainfall(T, MM, S_ID),
3    rainfall(T, MM_OTHER, S_ID_OTHER),
4    (S_ID != S_ID_OTHER),
5    (MM_OTHER != 0).
6
7  suspiciousRainfall(T, MM, S_ID) :=
8    rainfall(T, MM, S_ID),
9    (MM > 2),
10   not rainfallAtStationOtherThan(T, S_ID).
11
12 unsuspiciousRainfall(T, MM, S_ID) :=
13   rainfall(T, MM, S_ID),
14   (MM > 2),
15   not suspiciousRainfall(T, MM, S_ID).
16
17 unsuspiciousRainfall(T, MM, S_ID) :=
18   rainfall(T, MM, S_ID),
19   (MM <= 2).
```

Figure 2: Text-based logic programming example for finding suspicious rainfall sensor readings.

well-suited for capturing complex domain knowledge and for correlating data over time.

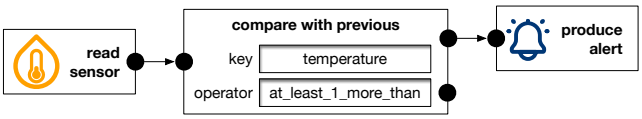
III. Problem Statement

The problem statement concerns five sub-problems. Two that emerge from using flow-based programming, and three that emerge from rule-based programming.

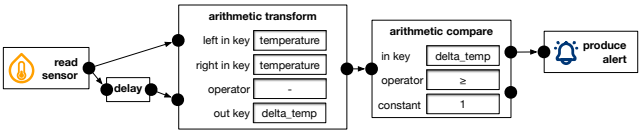
A. Obstacles to Process Sensor Data with Flow Operators

We have identified two problems that emerge when flow-based programming is used to build applications like those in the presented use cases.

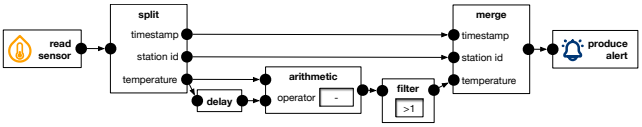
P1: Poor Visualisation of Correlation in Flows: The main strength of a flow-based environment is its ability to visualise control flow in an easy to grasp manner. However, they fall short in adequately visualising the dependencies in the context of data correlation, especially when that correlation concerns data that arrives over time. To exemplify this problem, we have made three sketches using flow-based abstractions for the identification of temperature spikes. Figure 3a presents a program that utilises custom flow component to correlate facts. These types of components hide the delay needed for time-based correlation in the implementation of the "compare with previous" component. I.e., there is no (visual) indication that the flow delays the processing of temperature readings. Figure 3b and Figure 3c use traditional flow-based components with a "delay" component that delays a reading for one time-step. In the former approach, an "arithmetic transform" component computes the difference between two temperature readings, and an "arithmetic compare" component then identifies the spikes. In the latter approach (based on [7]), individual temperature measurements are delayed and consequently compared to the current temperature reading. Note here that the semantics of the system needs to adequately handle missing values. For example, the "merge" component would need to discard data whenever the "filter" component dropped temperature readings.



(a) Using custom flow components.



(b) Using an explicit `delay` component and arithmetic components that operate on compound facts.



(c) Using an explicit `delay` component applied on an individual field. Comparisons are applied on values, not compound facts.

Figure 3: Three approaches for finding spikes in two consecutive temperature readings.

P2: Poor Abstraction of Sub-tasks in Flows: Many sensor data processing pipelines, such as those in the smart water management domain, mainly reason about compound facts, not individual numbers. In the application domain, typical input values are not merely primitive values like numbers or text, but compound objects or *facts* about the world that consist of multiple properties or *fields*. For instance, a temperature measurement of $25.0\,^{\circ}\mathrm{C}$ is not represented by a raw number "25.0", but by a "Temperature" fact that states that the temperature at some timestamp at some measuring station was "25.0". As a result, the processing pipelines have two layers: one layer dealing with high-level fact routing pipeline, and one layer implementing the multitude of lower-level fact transformation, filtering, and correlation tasks. Traditional flow-based approaches fail to offer low-code tools with which users can abstract over lower-level tasks (and their internal dependencies) in the implementation of the higher-level layer. This is exemplified by Figure 3c, in which individual fields are de-structured. The same visual language for considering whole facts is used on the value level, which makes understanding the flow at a glance more difficult.

B. Obstacles to Process Sensor Data with Rules

We have also identified three problems that emerge from using a rule-based approach in a low-code environment.

P3: Complexity of State Management in Rules: Many rule evaluation engines employ a stateful fixed-point evaluation model: i.e., facts are continuously added and derived throughout the lifetime of an application. While the incremental generation of facts is powerful, a program will eventually run out of memory. As such, there must be a mechanism to discard old facts. Automatic solutions to discard stale data from the knowledge base exist [8][9], though those solutions make assumptions about the facts' data model and about the constraints that the programmer specified in the rules which may not apply in general. This leaves state management a responsibility of the user of the rule engine. In the context of a low-code programming environment, we thus need to minimise the need for state management.

P4: Poor Fit of Rules to Imperative Actions: Rules are a poor fit for expressing imperative actions (e.g., network and file I/O). Imperative actions, like reading sensor data in UC1, do not map well onto the rule-based paradigm.

P5: Poor Modularity of Rules: Many rule-based engines, by default, operate on a single shared fact base for all rules: i.e., all rules are continuously active and operating on the same facts. In code-based approaches to rule-based programming, some advanced scoping mechanisms such as namespaces and modules [10] exist that can be used to prevent that programmers must take into account all combinations of all facts. However, requiring users to make use of such mechanisms as-is is at odds with the simplicity promised by low-code environments.

IV. Approach

We claim that when programming non-trivial applications, the programming model should enable programmers to express both types of logic in the corresponding paradigm. This is especially true in the context of (visual) low-code programming. If the goal of a visual low-code programming model is to be accessible by domain experts, then it is *essential* that both paradigms can be used for expressing programs.

A novel low-code platform that supports the vision outlined in this paper enables experts in the water management domain to express automatic data validation and processing pipelines, and in which the results of those pipelines can be used to make the (alerting) decisions needed to implement the scenarios.

Our solution is composed of a two-layered approach in which flow- and rule-based paradigms are integrated into one coherent low-code platform. The flow-based abstractions provide a clear, general, overview of the behaviour of the application. Complex rule-based abstractions are embedded within the flow and provide powerful abstractions for, e.g., aggregation which are not straightforward to express with pure flow-based abstractions. We now discuss various aspects of this programming environment, and how its design tackles the problems from Section III.

Rule-based Specification of Sub-tasks (P1 and P2): The main paradigm of the system is flow-based. However, complex processing steps can be implemented via rules components whose behaviour integrates well into the visual abstractions of the flow-based system. The system enforces a clear separation of fact types: different fact types are visually distinct in both the flow-based and rules-based programming environments. Edges between components in the visual environment are labelled by the type of facts to easily denote the flow of data, which makes flows easier to understand.

Rules components express application logic using "if-then" clauses. In short, the rule-based components provide users with the tools to abstract over lower-level sub-tasks inside the high-level, flow-based processing pipeline, while using a uniform

data language between the high-level (flow-based) abstractions and the low-level (rule-based) abstractions. Dealing with *correlation* of multiple facts is then expressed in this *lower* abstraction. I.e., instead of visualising correlation as a "delay" component (as in Figures 3b and 3c), the rules component as a whole reminds users that the block performs complex aggregation of data. We envision a block-based visualisation for the rule-based layer as those environments compare favourably with respect to flow-based environments [11].

Opt-in Statefulness (P3): Considering that rule-based approaches cannot completely forego state management, we instead let users explicitly choose either a stateful or a non-stateful execution model for each rule-based component. As such, the complexities of state management only need to be considered when users actually require statefulness. In short, we provide a (flow-based) component that contains a fully-fledged rule engine (with fixed-pointing semantics), and another (flow-based) component that applies simple transformations (in which correlations between facts are disallowed). Nonetheless, both use the same block-based visual language to remain accessible to domain experts.

Restricted Imperative Actions (P4): The visual platform uses the flow-based system for all imperative actions like reading data from a file, connecting to a sensor's live feed of measurements, and writing computed facts to a file. As such, these are not a concern in the rule-based programs at all. This simplifies not only the implementation of the rule-based engine, but also helps users to better understand the logic of a program.

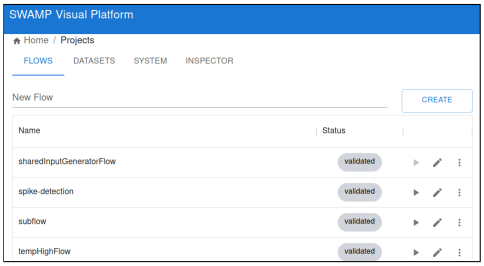
Modular Fact Bases for Rules (P5): Each rule-based component reasons about facts from only two origins: the extensional facts that were delivered to the rule-based component along the arrows visualised in the high-level flow program, and the intensional facts that were derived by the rules specified in that specific rule-based component, using only the facts that are local to that specific rule-based component. In both cases the rules of each rule-based component reason only about facts local to that flow component. This design element solves the poor modularity that follows from rules' whole-program scoping in an intuitive manner, linking it back to the way in which data dependencies at the level of facts are visualised in the flow language.

V. Bird's Eye Overview of the Platform

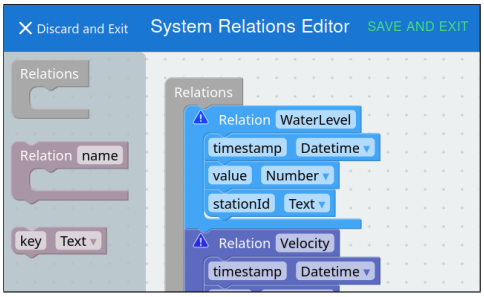
We now present the concrete aspects of the implementation of the platform. I.e., we present how users use this system to define and execute flows. The visual platform is implemented as a web application in which users can define flows and their respective data schemas.

A. Flows, Relations and Datasets

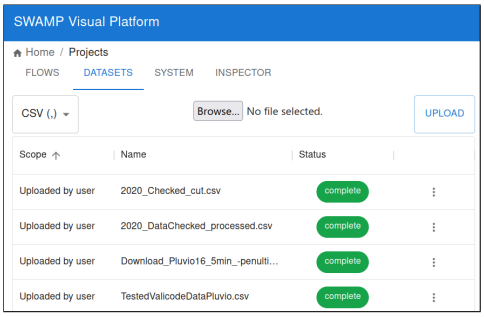
User interaction with flows starts from the main flow tab as shown in Figure 4a. On this tab users can manage flows. They can create a new flow, open the editor for any that have been created previously, as well as start/stop them.



(a) Managing flows



(b) Managing system-wide relations



(c) Managing datasets

Figure 4: Screenshots of the visual platform.

To distinguish between different fact types, the system allows for creating so-called "relations". The user can open the relation editor, which is shown in Figure 4b. These show the relations that are available to all flows. The same visual language for building rules is used, which we explain in Section V-C. By default, these relations include fact types relevant to the water management domain: i.e., timestamped measurements. As these are hardcoded by the system, they cannot be modified. However, new ones can be added and also modified from this interface.

Datasets are the platform's abstraction for persisted data. They are, in essence, CSV-files: i.e., an ordered collection of tuples with a certain arity. Datasets can be created by a flow component, or by uploading a CSV-file to the platform via the web interface as shown in Figure 4c. Datasets uploaded as a CSV-file can be loaded via a flow component. Note that datasets do not always correspond with a system-wide relation. As such, configuring a dataset component will automatically generate a relation in the flow where they are being used. To avoid the complexities that arise when multiple flows use the same dataset as input and/or output, only uploaded CSV-files

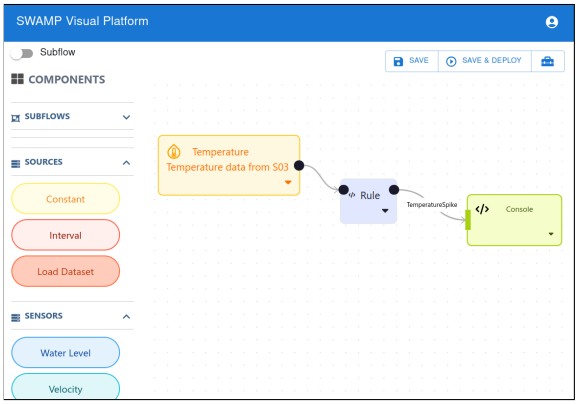


Figure 5: Screenshot of the flow canvas for detecting temperature spikes from a measurement station.

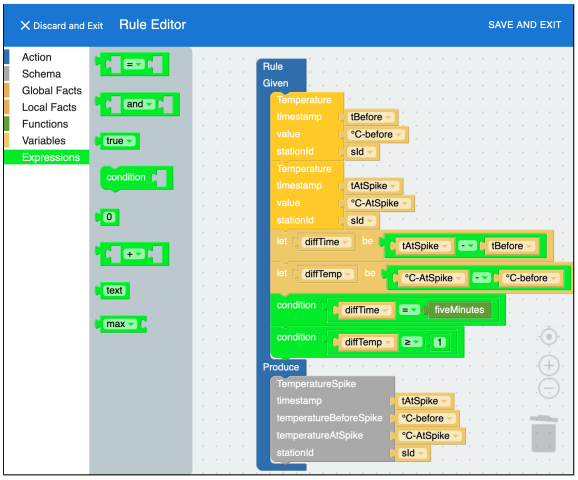


Figure 6: Screenshot of the visual editor for rules to detect temperature spikes.

can be used as input in a flow. This design restriction ensures that datasets cannot be used as an inter-flow communication mechanism, which is not supported by the platform.

B. Flow-Based Programming in RuleFlow

The nodes in a flow are components that produce, process, or consume facts. Using the terminology of [2], each component in a flow has zero or more "in ports", and zero or more "out ports". The type of in and out ports that a component has depends on the component's configuration. For instance, if a rule-based component is extended with a new fact pattern of a relation that it did not yet have before, the component will gain an in port for facts of that type.

Users add components to a flow by dragging a representation of one of the existing component kinds from a *component palette* onto the *flow canvas* (see Figure 5). Dependencies among components are established by dragging an arrow from a component's out port to another component's in port. These arrows are labelled with the name of the relation that travels along the arrows, and the arrows are colour-coded. This is similar to, e.g., DiscoPar [12]. Components can be configured by double-clicking on them, which usually opens a modal. Finally, the flow canvas offers the means to edit the global relations, offers buttons to save the state of a flow and to start the flow, and shows a console onto which the results of the flow (or of individual components) can be inspected. The platform is shipped with a number of built-in components, we distinguish between three main types.

1) *Source* components are used to provide input to a flow. There are built-in sources for generating facts with numeric values in a given range, for reading data from an existing dataset, and for connecting to a remote server to fetch (historic or live) data from measuring stations. The configuration of these measuring station components is kept simple: a user only needs to select from a list of measuring stations and the given date–time range for which measurements must be retrieved.
2) *Operator* components apply transformations. There are only two built-in operator nodes: one in which a rule-based program is embedded, and one which only applies simple transformations (as mentioned in Section IV, both use the same visual abstractions). The input and output ports are generated from the embedded rule-based program. For example, if there is a rule that expects "Rainfall" facts, then a "Rainfall" input port will be generated.
3) *Sink* components are the complement to source components. The built-in sink components are used to save facts to a dataset and to send data to the remote sensor platform (i.e., for UC2). There is also a built-in component that is used to send out email alerts (i.e., for UC1).

Besides these built-in components, users of the platform can define their own flows which then become available to other flows as components. Like operator components, *subflow* components can have both input and output ports. While a full overview of the semantics of subflows is not in the scope of this paper, the gist is that flows can be explicitly configured that they can be instantiated by other flows (i.e. top-left of Figure 5): the sources and sink components of these subflows are then parameterised.

C. Rule-Based Programming

Both operator components are configured in a rule-based programming environment, as depicted in Figure 6. Each rule-based subprogram consists of one or more rules, with one or more body fact patterns and one head fact pattern each. The environment is built on top of Blockly [13].

Blocks are provided for managing the data schema of facts (for any relations that are local to the rules component), and for defining rules using fact patterns. The fact patterns in the rules can make use of variable bindings, of an expression language, and of aggregators. In the visual language, the grammar is enforced by the shape of blocks' slots.

The block-based approach makes it possible to use a visual metaphor to denote that the `action` block "accepts" one `head fact`. We found that it was advantageous to use the shapes as blocks as a form of static "type" checking to prevent the construction of structurally wrong programs. On

the other hand, we found that disallowing connections due to more complex context-dependent requirements hampered users more than it helped them. Therefore, there are two types of blocks: *(1.)* blocks that deal with facts connect vertically to "notches", and *(2.)* blocks that deal with *fields* of facts connect horizontally to "jigsaw" slots.

When the visual platform detects that the user made a mistake that is not handled by this distinction, the platform allows the user to drop the block in the 'wrong' slot but provides inline feedback on why this connection does not make sense. This design choice serves two purposes: first, it enables the low-code platform to teach its users some of the finer nuances, and second, it allows users to construct programs which are not yet valid, but will become valid when the users finishes adding the blocks they meant to add. This is similar to how textual IDEs allow programmers to temporarily have a program in an invalid state while the programmer is halfway through making an edit.

D. Prototypical Implementation

A prototypical implementation of our approach was built in TypeScript. We leverage ReactFlow [14] for visualising and interacting with flows of components. The rule and expression language Rocks was built on top of Blockly [13]. Access to the platform was given to our research partner who experimented with designing surface water monitoring flows on the platform.

VI. Conclusion and Future Work

We described a low-code programming model and environment for automating sensor data processing pipelines for the surface water management domain. We identified visual flow-based programming and rule-based approaches as two promising avenues for building low-code programming models. Our prototypical platform has been designed specifically for the water management domain. We have shown, throughout the paper, the advantages that our platform provides for two use cases important to the water management domain. However, not all aspects of the problems identified in Section III are wholly resolved, and real-world use of the current design by its users point at avenues for future research.

A. Linking Flow Definition and Use

An important aspect for which our current design does not offer affordances, is the evolution of the low-code programs over time. Because of the way that the flow abstraction mechanism works, the site of use and the site of definition of a subflow are detached. The site of use does not track the site of definition.

B. Hot-swapping of Stateful Components

Evolving long running stateful software systems requires that care is taken to preserve accumulated state across modifications to the software. This holds equally in a flow-based low-code context, where modifying one or more flow components should not invalidate all state in the system. Further complicating the support for hot-swapping [15] is the fact that the state may need to be transformed to be compatible with the modified flow. For instance, if a user decides to merge two consecutive rules components in a flow together into one larger rules component, the system has to provide the means to correctly merge both components' fact bases.

References

[1] M. Hirzel, "Low-code programming models," *Commun. ACM*, vol. 66, no. 10, pp. 76–85, Sep. 2023, ISSN: 0001-0782. DOI: 10.1145/3587691.

[2] J. P. Morrison, "Flow-based programming," in *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, CreateSpace, 1994, pp. 25–29.

[3] M. Blackstock and R. Lea, "Toward a distributed data flow platform for the web of things (distributed node-red)," in *Proceedings of the 5th International Workshop on Web of Things, WoT 2014, Cambridge, MA, USA, October 8, 2014*, ACM, 2014, pp. 34–39. DOI: 10.1145/2684432.2684439.

[4] The NoFlo Team, *Noflo: Flow-based programming for javascript*, (accessed: 03.11.2023).

[5] M. Burnett, C. Cook, and G. Rothermel, "End-user software engineering," *Commun. ACM*, vol. 47, no. 9, pp. 53–58, Sep. 2004, ISSN: 0001-0782. DOI: 10.1145/1015864.1015889.

[6] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989. DOI: 10.1109/69.43410.

[7] A. Catalá, P. Pons, J. J. Martínez, J. A. Mocholí, and E. Navarro, "A meta-model for dataflow-based rules in smart environments: Evaluating user comprehension and performance," *Sci. Comput. Program.*, vol. 78, no. 10, pp. 1930–1950, 2013. DOI: 10.1016/J.SCICO.2012.06.010.

[8] T. Renaux, "A distributed logic reactive programming model," English, ISBN 978-9-49307-920-5, Ph.D. dissertation, Vrije Universiteit Brussel, 2019, ISBN: 978-9-49307-920-5.

[9] D. Teodosiu and G. Pollak, "Discarding unused temporal information in a production system," in *Proc. of the ISMM International Conference on Information and Knowledge Management CIKM-92*, Citeseer, Baltimore, MD, 1992, pp. 177–184.

[10] L. Sterling and E. Y. Shapiro, "The art of prolog: Advanced programming techniques," in MIT press, 1994, pp. 243–244.

[11] D. Mason and K. Dave, "Block-based versus flow-based programming for naive programmers," *IEEE Blocks and Beyond Workshop (B&B)*, pp. 25–28, Oct. 2017. DOI: 10 . 1109 / BLOCKS.2017.8120405.

[12] J. Zaman, K. Kambona, and W. De Meuter, "DISCOPAR: A visual reactive programming language for generating cloud-based participatory sensing platforms," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2018, Boston, MA, USA: Association for Computing Machinery, 2018, pp. 31–40, ISBN: 9781450360708. DOI: 10 . 1145 / 3281278.3281285.

[13] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with blockly," in *2017 IEEE blocks and beyond workshop (B&B)*, IEEE, 2017, pp. 21–24.

[14] xyflow Team, *React flow - a library for building node-based uis*, https://reactflow.dev/; [retrieved: February, 2025], 2024.

[15] L. Vanbever, J. Reich, T. Benson, N. Foster, and J. Rexford, "Hotswap: Correct and efficient controller upgrades for software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 133–138, ISBN: 9781450321785. DOI: 10.1145/2491185.2491194.