A Multi-Paradigm Modelling approach to live modelling

SoSyM manuscript No.
(will be inserted by the editor)

A Multi-Paradigm Modelling Approach to Live Modelling

Yentl Van Tendeloo · Simon Van Mierlo · Hans Vangheluwe

Received: date / Accepted: date

Abstract To develop complex systems and tackle their inherent complexity, (executable) modelling takes a prominent role in the development cycle. But whereas good tool support exists for programming, tools for executable modelling have not yet reached the same level of functionality and maturity. In particular, live programming is seeing increasing support in programming tools, allowing users to dynamically change the source code of a running application. This significantly reduces the edit-compile-debug cycle and grants the ability to gauge the effect of code changes instantly, aiding in debugging and code comprehension in general. In the modelling domain, however, live modelling only has limited support for a few formalisms. In this paper we propose a multi-paradigm modelling approach to add liveness to modelling languages in a generic way, which is reusable across multiple formalisms. Live programming concepts and techniques are transposed to (domain-specific) executable modelling languages, clearly distinguishing between generic and language-specific concepts. To evaluate our approach, live modelling is implemented for three modelling languages, for which the implementation of liveness substantially differs. For all three cases, the exact same structured process was used to enable live modelling, which only required a "sanitization" operation to be defined.

Y. Van Tendeloo
University of Antwerp, Belgium
E-mail: Yentl.VanTendeloo@uantwerpen.be

S. Van Mierlo
University of Antwerp, Belgium
E-mail: Simon.VanMierlo@uantwerpen.be

H. Vangheluwe
University of Antwerp, Belgium;
Flanders Make vzw, Belgium;
McGill University, Montréal, Canada
E-mail: hv@cs.mcgill.ca

Keywords Live programming · Live modelling · Debugging · Multi-Paradigm Modelling

1 Introduction

Complex software-intensive systems are becoming more and more pervasive in our daily lives [32]. Modelling, and in particular domain-specific modelling [18] (DSM), has proven to be an essential technique to avoid the accidental complexity incurred when using traditional programming techniques, by allowing domain experts to specify systems in a notation they are familiar with. Historically, models were mostly used for documentation. More recently, they are increasingly used for execution, for example through code generation [18] or simulation/interpretation, often implemented using model transformation [39]. This brings forth the need for debugging the execution, as seen in the programming community. While there is a growing interest in model verification, not all models can be verified due to the size of the state space, or due to lacking (efficient) tool support. Furthermore, model verification can indeed aid in finding whether a system is correct, but it is often unable to track down the source of the violation. As such, debugging will remain a vital part of the modelling process.

Commercial modelling and simulation tools often provide limited support for debugging. The research community has recently made several contributions that enable specific debugging features for several types of formalisms [22,8, 34,51,2]. Compared to code debugging, which has a wide variety of debugging operations [52], current modelling approaches and tools supporting them are still in their infancy in terms of features, applicability, and usability. It is therefore tempting for system developers to debug the automatically generated code directly, instead of the model itself [10].

Live programming [43] is an advanced feature of several programming tools, allowing programmers to modify the code of applications while the application is running, immediately having the new code integrated in the *running* application. There is no apparent compile and re-run cycle, reducing the cognitive gap between program code and execution. Additionally, the state of the running application is transparently transferred from the running program to the newly compiled version of the code, thereby, removing the need to redo all operations up to the point in time where the change was made. While there are already several tools that support live programming, making a programming language "live" is done ad-hoc, and is refered to as a black art [5]. As such, it is difficult to transpose liveness techniques between languages.

In this paper, we transpose the essence of live programming to the modelling domain, in a generic way. Contrary to live programming, where only a single language is considered most of the time, domain-specific modelling raises the need for many different domain-specific formalisms. Many of these domain-specific formalisms only have a handful of users, rendering the investment for implementing live modelling techniques in an ad-hoc way difficult to justify. Therefore, live modelling should be implemented in a generic way, making it applicable to many modelling formalisms with minimal effort. Support for live modelling was identified as a key feature to advance the usability of model-driven techniques [19]. The research question thus is "how can live programming concepts be ported to the modelling domain, making them generically applicable". Despite mostly being presented as a debugging operation in this paper, live formalisms can be applied to other situations as well, such as education or model comprehension in general.

To effectively support live modelling in the context of domain-specific formalisms, we deconstruct the traditional live programming process, and reconstruct it in the context of modelling by applying concepts and techniques from live programming to executable modelling formalisms. All activities related to liveness are distilled into a single operation, which we term *sanitization*. To make a formalism live, only the sanitization operation should have to be updated, while reusing all other aspects of live modelling. Note that liveness only applies to executable modelling formalisms, and we therefore limit ourselves to these in this paper.

We present a Multi-Paradigm Modelling (MPM) [49, 30] approach to live modelling. MPM advocates the *explicit* modelling of all pertinent parts and aspects of complex systems. It adresses and integrates three orthogonal dimensions: (1) multi-abstraction modelling, concerned with the relationships between models (e.g., refinement and generalization); (2) multi-formalism modelling, concerned with the coupling of and transformation between models described in different formalisms (e.g., multi-view and multi-component); and (3) explicitly modelling the often complex, concurrent workflows. We present an explicitly modelled framework for the definition of live modelling formalisms: all aspects of the approach, including the process [23], are explicitly modelled. Our approach therefore relies on MPM tool support. Additionally, our approach is especially useful in the context of MPM, where various domain-specific formalisms are used and processes can be enacted.

We distinguish between three types of executable modelling formalisms for which the implementation of the sanitization operation is fundamentally different. For each, we present a representative example, which we use throughout this paper as a running example: Finite State Automata (FSAs) [17], Discrete Time Causal Block Diagrams (DTCBDs) [7], and Continuous Time Causal Block Diagrams (CTCBDs) [7].

These modelling formalisms, and their live implementation, are implemented in the Modelverse [47], our Multi-Paradigm Modelling environment [48]. The Modelverse provides full support for Multi-Paradigm Modelling, thereby providing the necessary tool support for language engineering, model transformations, process enactment, and the use of multiple flexible interfaces. All code is available online.[1]

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on live programming and executable modelling. Section 3 presents three running examples: a Finite State Automata model, a Discrete Time Causal Block Diagrams model, and a Continuous Time Causal Block Diagrams model. Section 4 presents our approach to live modelling, explaining our concepts and method, and demonstrates its application to our running examples. Section 5 presents a prototype implementation of our approach for the three examples. Section 6 presents related work, and Section 7 concludes the paper.

2 Background

As this paper combines techniques from the live programming and executable modelling domain, both domains are first briefly introduced.

2.1 Live Programming

Live, or interactive programming aims to bridge the "gulf of evaluation" [21,42]. It allows users to update the source code of an application while it is running, with changes being applied instantly in the running application. There is therefore no need to manually recompile, restart, and rerun the program up to the point of execution when the modification was made. This has several advantages, such as decreasing the length of

[1] `https://msdl.uantwerpen.be/git/yentl/modelverse`

the edit-compile-debug cycle, and offering users immediate insight in the effect of code changes. An example of live programming, as implemented by ElmScript [9], can be seen online.[2]

Basically, the process of live programming is as follows.

1. A developer writes code in a programming language.
2. The (valid) code is compiled to instructions for the specific machine.
3. The instructions are loaded into memory, and storage is allocated for execution.
4. The program is executed, which performs operations on the program and its state.
5. The developer modifies the code of the program, concurrently with execution.
6. The modified code is compiled to new instructions.
7. The program merges its old instructions and state with the new instructions.
8. The program executes the new instructions.

With the exception of the 7th item, these steps are identical to the workflow of normal programming. Normally, however, the new instructions are only executed in a new invocation of the program. The merge operation, therefore, is the only new operation in live programming (from a functional point of view). The merge operation alters a running program to incorporate changes unknown at compilation time, by merging the updated set of instructions with the old state of the running program. Specifically, new instructions that do not have an execution context are merged with old instructions and their associated execution state. As data is also merged, such as the value of variables, information from the old program must be combined with the new instructions.

Data merging is intentionally left vague, as many approaches exist. Three categories were proposed [26], depending on how much data is copied: no live programming, recorded event, and real-time. We illustrate all three with a game example, similar to the ElmScript example. The game is a simple platform game, where the jump height of the character is updated during execution. The game's current state is shown in Figure 1a, where the character jumped onto the platform and, in the meantime, collected one coin. If the character were to jump, the coin is collected and the score is increased to 2.

No live programming is the most basic, where no information is passed between executions. Upon recompilation, the currently running application is terminated and restarted afresh. This approach does not implement live programming at all, and can easily be replicated without any modification to the programming language itself. All that is required is an automatic restart of the application after a change is detected. In the game example, the character is respawned at the beginning and the score is initialized to zero. This is shown in Figure 1b, where the character has respawned and all coins have been reset as well. From this point onwards, the jump height is reduced and the character will be unable to jump on the platform. In conclusion, no state is retained.

Recorded event takes over the history of all inputs sent to the old running application. The new program is then executed with these simulated events, making it seem as if the inputs sent to the old program were sent to the new program. This approach is used in programming languages such as ElmScript [9]. For performance reasons, the program is often not completely re-executed, but only dependent functions are re-evaluated. In the game example, our character might switch location and score, depending on what these values would be if the exact same inputs were given in the new application. When the jump height parameter is decreased, we suddenly find the character below the platform, instead of on top of it. This is shown in Figure 1c, where we see the character below the platform: the jump we did before did not reach the same height, which made the character unable to reach the platform. Subsequent actions, such as moving to the right, were still replicated, but in a different context: below instead of on top of the platform. In conclusion, the input history part of the state is retained.

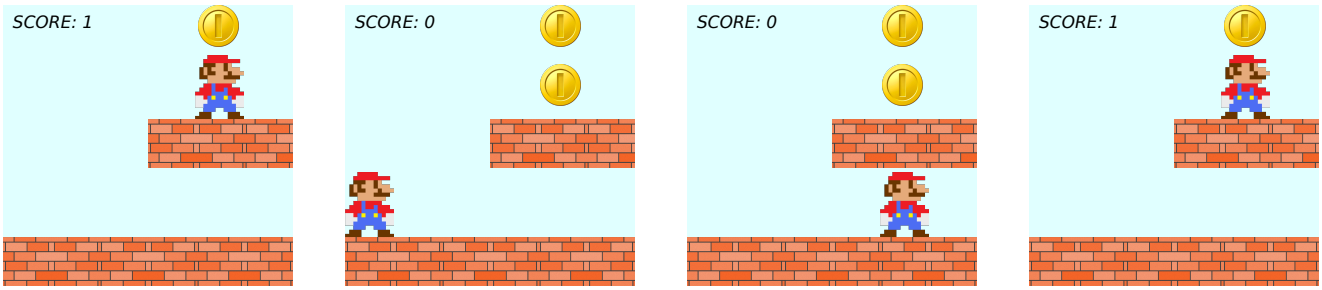
Real-time takes over the complete history of the old running program, but merges in new instructions to be used in the future. The new program is effectively a rewritten version of the old program, which just continues computation. This approach is used in programming languages such as Smalltalk [14], and is often also termed *fix and continue*. In the game example, our character will be at the same location and have the same score as before, but changes will take effect from that point onwards. When the jump height parameter is decreased, we find it impossible to jump as high as we could before, though our current location remains unchanged. This is shown in Figure 1d, where we see no immediate change. From this point onwards, however, we are unable to get the coin right above us, as the character can no longer jump that high. In conclusion, the complete state is retained.

In the remainder of this paper, we will mostly consider *real-time* live programming, as this was previously identified as being the most appropriate for simulation [26].

2.2 Executable Modelling

Modelling has historically mostly been used in the form of documentation of a separate coded application. Recently, however, executable modelling has gained popularity, where the model itself becomes the final application, without additional coding effort. In this case, the model is not necessarily used as documentation or to generate skeleton code, but its execution becomes detached from programming. In essence, models have gained semantics, for which two main categories exist: denotational and operational semantics.

[2] `http://debug.elm-lang.org/`



(a) Original configuration. (b) No live programming. (c) Recorded event. (d) Real-time.

Fig. 1: State of the game before and after decreasing the jump height parameter.

Denotational semantics, or translational semantics, consists of mapping the model to a different formalism, for which a semantics does exist. For example, if the model is translated to programming code, this is denotational semantics. The semantics is then only concerned with the translation in itself: generating code. How this code is executed, is beyond the scope of denotational semantics. Denotational semantics is mostly used to employ existing knowledge and formalisms. For example, it is popular to generate code (leveraging existing code execution platforms) or map it to different domain-specific formalisms, such as Petri nets [31] (leveraging existing analysis methods). Chaining is possible, where the target formalism has denotational semantics itself, as long as eventually we end up with a formalism that has operational semantics defined.

Operational semantics defines the semantics by executing the model directly: the model is transformed from one valid configuration to another. For example, a *current state* flag is kept for all elements, which is moved along the model to indicate the currently executing state. Contrary to denotational semantics, operational semantics does not rely on other formalisms, but it requires additional data to be stored somewhere. For this purpose, a distinction is often made between a design and runtime metamodel.

The design metamodel is the metamodel that is used by the designer when creating the model. It has all the necessary constructs for design, but is not concerned with the execution. For example, in Finite State Automata (FSAs), a State only has a *name* and *initial* attribute. The runtime metamodel, however, has additional information required for execution. For example, the state now still has its *name* and *initial* attribute, but additionally has a *current* attribute. This attribute stores a boolean containing whether or not this is the current state of the execution. While this information is required for the execution, as it needs to be stored somewhere, it is invisible to the designer.

Due to the distinction between these two types of metamodels, multiple models are actually required for operational semantics: the design model is first translated to a runtime model, thereby initializing it (e.g., setting the *current* attribute to the *initial* attribute). The actual operational semantics is subsequently executed on this intermediate runtime model.

3 Running Examples

In this section, we present the formalisms used as running example throughout the remainder of the paper. For all formalisms, we present a simple model on which we use live modelling. Modelling formalisms can have widely varying semantics, including non-determinism, event-driven behaviour, timing, etc. While implementing live modelling techniques for each of these categories will be different, one essential difference that has an effect on live modelling is the types of changes that can be made. We identify three types of formalisms: two that gain semantics through operational semantics (i.e., they manage the state themselves), with support for breaking and non-breaking changes, and one that gains semantics through denotational semantics (i.e., it delegates execution and states to another formalism). The distinction between breaking and non-breaking changes stems from the language evolution community [28], where changes to the metamodel can be considered to break the instances. With breaking changes, the instances have to be adapted, for example when adding a new mandatory attribute to a class. This can be either resolvable (e.g., when the attribute has a default value) or non-resolvable (e.g., when the attribute has no default value). With non-breaking changes, the instances do not have to be adapted, for example when adding an optional attribute to a class.

With *operational semantics and breaking changes*, conforming changes on the design model might result in non-conforming changes on the runtime model. For example, in live programming, the currently executing line of code can be removed. In this case, the change in the design model (source code) is a valid piece of program code, but when this same change is mapped to the runtime model, the current instruction pointer is also removed, making the runtime model invalid. To solve inconsistent states after such a change, a new line of code must be selected as the currently execut-

ing line of code, which can often not be done automatically. As a representative example of a formalism with breaking changes, we choose Finite State Automata.

With *operational semantics and non-breaking changes*, conforming changes on the design model always result in conforming changes on the runtime model. For example, in live programming, variables can be added or removed, and their values cannot be changed. In this case, the change in the design model (source code) is a valid piece of program code, and when the same changes are mapped to the runtime model, the runtime model stays valid. Note that it is not possible to change the values of variables themselves, as we only have access to the source code, not to the execution information. As a representative example of a formalism with non-breaking changes, we choose Discrete Time Causal Block Diagrams.

With *denotational semantics*, execution is delegated to another formalism. For example, in live programming, the codebase can be first translated to another programming formalism, for which live programming is supported. To solve the inconsistent state, the functionalities of the target formalism can be used as-is. In the end, however, the modeller is likely only an expert in the source formalism, and might even be unaware of the existence of a target formalism. As a representative example of a formalism with denotational semantics, we choose Continuous Time Causal Block Diagrams, which we map to Discrete Time Causal Block Diagrams through discretization and optimization.

For readability, we present our approach using these three different formalisms. Of course, our approach is applicable for other formalisms as well. Many formalisms support different types of changes, such as some that are breaking (*e.g.*, the executing line of code) and others that are non-breaking (*e.g.*, variable values). Therefore, a composite merge rule is often required, which handles all aspects simultaneously. We explain all used formalisms next, along with an example model.

These three types of formalisms are exhaustive: a change is either breaking (i.e., requires changes to be applied to the model) or non-breaking (i.e., the model can be used as-is). For breaking changes, two resolution methods exist, both of which are handled in this paper. Denotational semantics does not consider the difference between breaking and non-breaking changes, as it merely relies on the underlying semantics for this. As such, denotational semantics is only considered in combination with one type of changes in this paper.

Note that we only consider the feasibility and general structure of live modelling for these formalisms. Different applications naturally raise new challenges. For example, one of the major challenges in live programming is efficient recompilation, as this takes a significant amount of time.

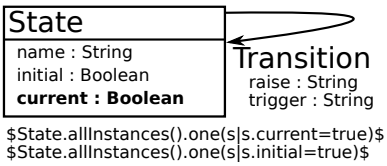


Fig. 2: Abstract Syntax of Finite State Automata. Runtime-only concepts are shown in bold.

3.1 Finite State Automata

The Finite State Automata (FSA) formalism [17] is used to model reactive systems with discrete state. Its building blocks are:

- *States*, which represent the state a system is in. There is exactly one initial state, where execution starts.
- *Transitions* between states that model the flow of the system. A transition is triggered by an event from the environment, consuming it as the transition is taken. Only transitions whose source state is the current state can fire. After triggering, the target of the transition becomes the new current state. A transition can additionally raise an output event to the environment.

Its abstract syntax is shown in Figure 2. Note the notation for *Transition*, which is an association going from a *State* to another *State*, having two attributes: a *raise* and *trigger* string. A pseudo-code version of its semantics is shown in Algorithm 1.

Algorithm 1 FSA operational semantics.

```
function SimFSA(M)
    state ← InitialStates(M).pickOne()
    while state ∉ FinalStates(M) do
        wait for input
        state ← Target(M, input)
    end while
end function
```

A frequently used visualization is as a state diagram, where states are represented as circles and transitions as arrows, labelled with their trigger and output event. The initial state is pointed to by an arrow starting from a small black dot. An example is shown in Figure 3, where a simple home security alarm system is modelled. In the *idle* mode, the alarm system can be armed by the user. If someone is detected in the *armed* mode, the alarm goes off, until the user inputs the correct combination. The alarm can be disabled by sending the *Disable* event, but only when no intrusion is detected.

The FSA formalism is an example formalism with potential breaking changes: the only state of the model is the current state, which is explicitly present and can thus be removed. If the user deletes the current state, execution can only

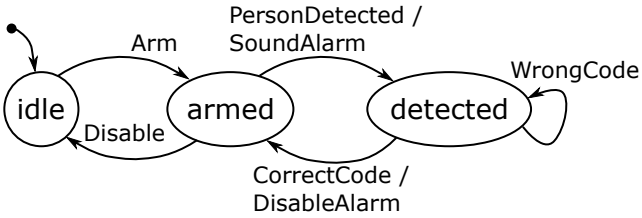


Fig. 3: Example FSA of a home security alarm system.

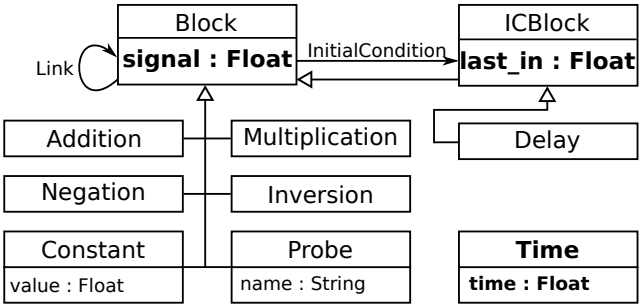


Fig. 4: Abstract Syntax of Discrete Time Causal Block Diagrams. Runtime-only concepts are shown in bold.

resume when another state is chosen as the current state. This can be resolved either manually (breaking, non-resolvable) or automatically (breaking, resolvable).

3.2 Discrete Time Causal-Block Diagrams

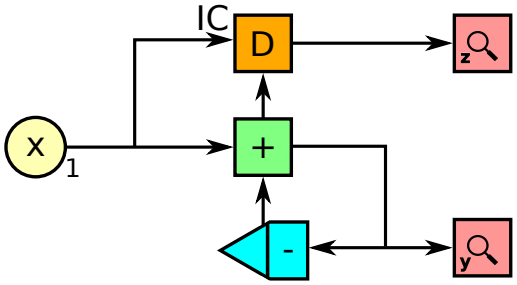
The Discrete Time Causal Block Diagrams (DTCBD) formalism [7] is a dataflow formalism, where signals are propagated through a network of connected blocks. It allows to model systems by defining them as a set of equations. The semantics is given by a set of continuous signals. Blocks implement atomic mathematical operations, which take their input signals and generate, instantaneously, a single output value. The mathematical concepts modelled by these blocks include constants, addition, negation, multiplication, and inversion. Additionally, a delay block is provided which holds the value for a single iteration, thus introducing the notion of "next step". At initialization, the block uses the value coming into it via the Initial Condition (IC) port. Connections between blocks indicate dependencies: the output of the source block is used as input by the target block.

Its abstract syntax is shown in Figure 4. Pseudo-code for its operational semantics is shown in Algorithm 2.

Figure 5a presents a simple DTCBD model representing the equations shown in Figure 5b. The equation for y is reduced to $y = x - y$, which is a direct feedback loop (termed "algebraic loop"). While this seems a trivial model to map to code, this is not the case: the algebraic loop must be resolved first. Indeed, in code, the statement `y = x - y` translates to the equation $y(t) = x(t-1) - y(t-1)$, as the old values

Algorithm 2 DTCBD operational semantics.

```
function SimulateCBD(M, maxIters, Δt)
    clock ← 0
    state ← InitSignals(M)
    numIters ← 0
    while numIters < maxIters do
        g ← DepGraph(M, numIters)
        s ← LoopDetect(g)
        for c in s do
            if c = {gblock} then
                state ← CompB(c, state)
            else
                state ← CompL(c, state)
            end if
        end for
        clock ← clock + Δt
        numIters ← numIters + 1
    end while
    return clock, state
end function
```



(a) Example DTCBD, containing an algebraic loop.

$$\begin{cases} y(t) = x(t) - y(t) \\ z(t) = \begin{cases} x(t) & \text{if } t = 0 \\ y(t-1) & \text{if } t > 0 \end{cases} \end{cases}$$

(b) The equations represented by the example DTCBD model.

Fig. 5: Example DTCBD.

of x and y are used. To actually implement the equation $y = x - y$, the algebraic loop must be solved to $y = \frac{x}{2}$, which can be implemented in code. Programmers therefore have to manually solve the set of linear equations to come up with the code to solve this system of equations. Small changes in the system of equations can result in large changes on the resulting solution.

In contrast, DTCBDs handle linear algebraic loops natively, solving $y = x - y$ automatically and generating the necessary code. To solve linear algebraic loops, the loop is detected as a strongly connected component, and a linear system of equations is constructed. In the case of Figure 5a, for example, we construct the set of equations shown in Figure 5b. These equations are automatically solved and code can be generated.

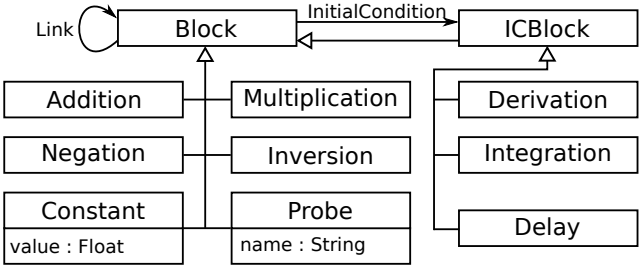


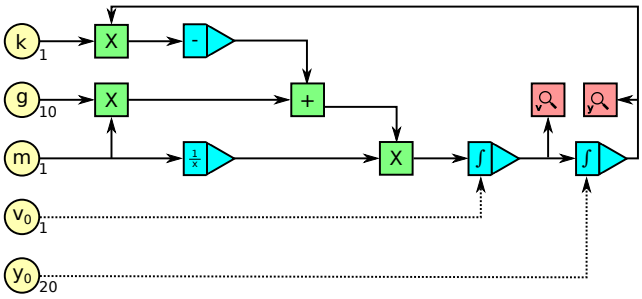
Fig. 6: Abstract Syntax of Continuous Time Causal Block Diagrams.

The DTCBD formalism is an example of a formalism with exclusively non-breaking changes: delay blocks have a memory of their previous iteration, but this is no longer necessary when the block is deleted. The runtime state of the model is an aggregation of the memory values, which the user cannot manipulate directly. When a delay block is added, the state needs to be updated accordingly by initializing a new state variable. Similarly, when a delay block is removed, part of the state is removed. It is impossible for a conforming design model to result in a non-conforming runtime model.

3.3 Continuous Time Causal-Block Diagrams

The Continuous Time Causal Block Diagrams (CTCBD) formalism [7] is an extension to DTCBDs, introducing two continuous blocks: an integrator and derivator. Its abstract syntax is shown in Figure 6. As denotational semantics is used, no runtime-only elements are present in the abstract syntax. While intuitively this could be implemented by extending the operational semantics of DTCBDs as well, this has several disadvantages. First, the operational semantics would have to be mostly duplicated, as it is mostly identical (e.g., topological sorting, algebraic loop detection, and iteration), meaning that there is code duplication, resulting in poor maintainability. Second, by mapping to DTCBDs, all operations on DTCBDs can be reused, such as live modelling, but also other techniques such as debugging. Third, the algebraic loop detection algorithm would have to be expanded as well, as algebraic loops can also exist in CTCBDs, where an integrator or derivator is part of the loop. By mapping them to DTCBDs, all algebraic loop detection and resolution algorithms can be reused as-is, without further additions.

Figure 7a presents a simple CTCBD model representing the equations shown in Figure 7b. These equations model the behaviour of a mass attached to a spring, which is going up and down. Most important is the addition of the integrator blocks, which were not possible in DTCBDs. When mapping this CTCBD to a DTCBD, the integrator blocks are expanded to a discretized version of the integrator, for example using the forward Euler approach. While both formalisms



(a) Example CTCBD model.

$$\begin{cases} v(t) = \int_0^t \frac{m \cdot g - k \cdot y(t)}{m} dt \\ y(t) = \int_0^t v(t) dt \end{cases}$$

(b) The equations represented by the example CTCBD model.

Fig. 7: Example CTCBD.

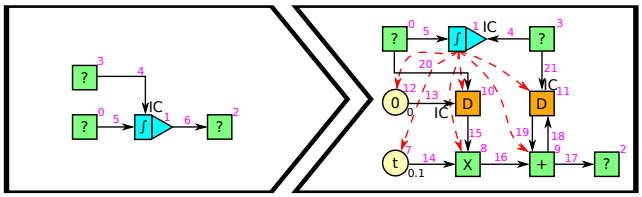


Fig. 8: Expand rule for an integrator block.

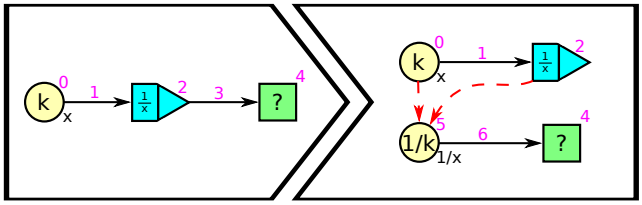


Fig. 9: Optimize rule for an invertor block.

look similar, there is a non-trivial translation step between them: discretization. Additionally, while discretizing, it is possible to perform an optimization step for constant folding, dead-block removal, and flattening [33]. Two example model transformation rules are shown for expansion (Figure 8, mapping 1 CTCBD element to multiple DTCBD elements) and optimization (Figure 9, mapping multiple CTCBD elements to one DTCBD element).

The CTCBD formalism is an example with denotational semantics: to execute the model, it is first translated to an equivalent DTCBD (with respect to some properties), which is then executed instead. It does not matter whether the target formalism has breaking or non-breaking changes, or has denotational semantics itself, as live modelling is assumed to be supported for that formalism already. As such, we build on top of the live modelling functionality that was developed for our other running example.

While we acknowledge that DTCBDs and CTCDBDs look similar, there is a non-trivial n-to-n mapping between both formalisms. Even though many concepts can be reused between the two, the mapping exhibits most of the complexities normally associated with traceability links in denotational semantics.

4 Live Modelling

We start our approach to live modelling by deconstructing the current process for live programming schematically, and then generalize the concepts and processes to modelling. This results in a general framework for live modelling, that can be applied to any (domain-specific) modelling formalism. Programming languages also fit this framework, as they can themselves be seen as an executable modelling formalism.

4.1 Deconstructing Live Programming

The first step in our work is the deconstruction of the live programming process. This process consists of artefacts (*i.e.*, files or structures in memory) and modifications (*i.e.*, operations on these artefacts). An overview is shown in Figure 10.

4.1.1 Artefacts

We distinguish three artefacts: code, instructions, and the running program.

The **code** is the textual notation that represents a program, created by the developer. Code is often persisted as a text file. It is the only artefact programmers should edit; they should not edit any subsequent (automatically generated) artefacts. An example is a *C++* source code file.

The **instructions** are the result of compiling the code. Consisting of a set of instructions and data, which can be interpreted by the machine. Execution-time concepts are not yet considered: variables have no value, nor is there a currently executing line of code. The compiled program is only an "intermediate" form: it is an optimized version of the original code, and is easier to read for a computer. As part of the compilation process, the program is instrumented with extra information, such as mapping variables to registers. An example is a compiled *C++* program in *ELF* format. It is important to note that these instructions are semantically equivalent to the original code.

The **running program** is the actual program loaded in memory, including its state. It is executed by the machine and is very similar to the compiled program, but it includes runtime information (the state). Multiple versions of the same program can execute at the same time without sharing state (*i.e.*, memory): each program runs independently of the others. Even when the instructions are changed (*i.e.*, in self-modifying code), these changes only take effect on the running instance. Thus, program execution can be defined as the continuous updating of the artefact itself. An example is the memory used for executing an *ELF* file, encompassing both the instructions and the execution data.

4.1.2 Operations

We distinguish five operations between these artefacts: compilation, initialization, execution, modification, and merging.

Compilation (code to instructions) transforms a human-readable piece of code to a machine-readable representation. This process involves steps such as making implementation decisions and register allocation. The generated machine code remains semantically equivalent to the original code.

Initialization (instructions to running program) loads a compiled program into memory and initializes its state at the start of execution. Apart from initializing the state, the machine code is copied to memory.

Execution (modification of running program) modifies the program by changing the data, or by changing the instructions (self-modifying programs). Execution typically only alters the state of the variables contained in the program.

Modification (modification of code) represents the changes a user makes to the original source code artefact. Arbitrary changes are supported, as long as the result is still a valid instance of the original language (*i.e.*, it can be compiled).

Merging (instructions and running program to a running program) merges the state of a running program with an updated set of instructions. The merge operation is specific to live programming: the currently executed program is merged with the updated instructions. Afterwards, the "new" program resumes execution where the "old" program left off, thereby replacing it. This can be seen as a generalization of the initialization operation: as part of the merge, the state is initialized for new instructions, while it is modified if instructions are removed or updated. We therefore consider initialization a merge with an "empty program".

The live programming process is shown in Figure 10, where we explicitly mention the type of artefacts for a specific scenario. That way, the signature of the operations becomes apparent. While live programming environments often offer additional features for performance reasons, such as incremental compilation, these are not functionally required.

4.2 Transposition to Modelling

Taking this diagrammatic process, we generalize to the domain of modelling. We port these concepts to the modelling domain: instead of using programming languages and execution on actual machines, we make it platform-independent. Whereas we used a language such as *C++* before, we now assume the artefacts as instances of a formalism. Our approach

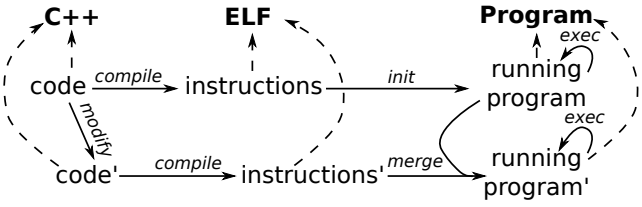


Fig. 10: Diagrammatic overview of live programming. Full lines represent operations, dotted lines represent typing relations.

is a generalization: it can also be applied to programming languages, since they can be seen as a formalism. Their syntax is defined in the language's grammar (cf. metamodel), while their semantics is defined by their mapping onto machine code.

4.2.1 Artefacts

First, we transpose the artefacts, which gives us three kinds of models: the design model (code), partial runtime model (instructions), and full runtime model (running program).

The **Design Model** is the equivalent of the *code*. Similar to code, it is the only artefact that the user can edit, and thus also the one that is seen as the "master" copy of the program. Our previous examples of an FSA, DTCBD, and CTCBD model, presented in Figure 3, Figure 5a, and Figure 7a, respectively, are expressed in the design language.

The **Partial Runtime Model** is the equivalent of the *instructions*. Similar to instructions, it has the same meaning as the design model, though it might be pre-processed. If operational semantics is defined for this formalism directly, it can be seen as a retyping operation. In general, however, the structure of both might vary significantly (as was the case with *C++* and *ELF*). In the FSA and DTCBD formalisms, the partial runtime models are equivalent to the design models, since both formalisms have operational semantics. In the CTCBD formalism, the partial runtime model differs, as it is a model in the target formalism: DTCBDs. Figure 11 presents a discretized version of the original CTCBD model, in the DTCBD formalism.

The **(Full) Runtime Model** is the equivalent of the *running program*. Similar to the running program, the full runtime model is a copy of the partial runtime model, extended with additional elements representing the execution state. In Figure 12, the full runtime models of the running examples are shown.

For FSAs (Figure 12a), a pointer to the *current state* is added. In the figure, the model is currently in the *detected* state. For execution, the model is updated by changing the current state based on the input events received from the environment.

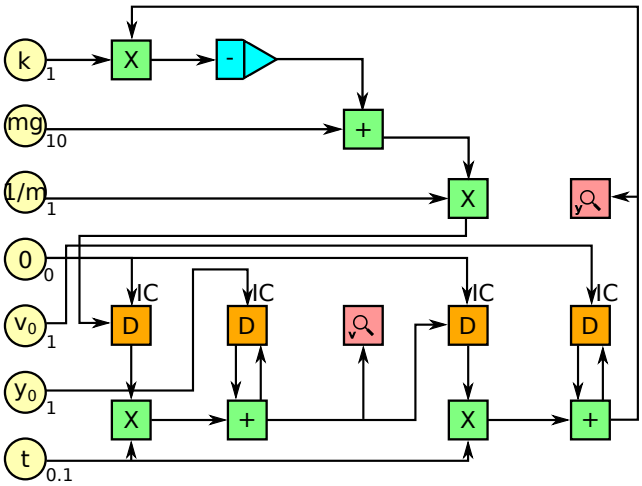


Fig. 11: The partial runtime model of the example CTCBD, as an instance of the DTCBD formalism.

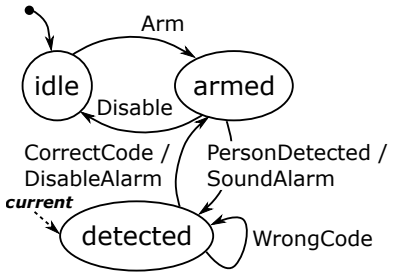
For DTCBDs (Figure 12b), more runtime information is added, as they have a notion of time, represented by the number of iterations. The *time* is incremented each time an iteration is executed. Each iteration, the signal values are (re)computed based on the new input values. For most blocks, their output signal value only depends on their current input values and hence they are stateless. One exception is the delay block, whose output value depends on its input value in the previous iteration. A *mem* runtime variable keeps track of this value, which must be initialized as well.

For CTCBDs (Figure 12c), the situation is identical to DTCBDs now, as the model was effectively translated to the DTCBD domain.

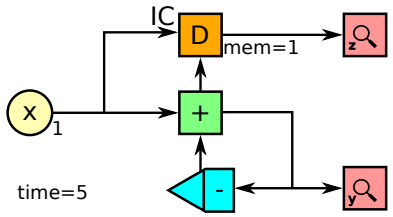
4.2.2 Operations

Second, we transpose the various operations on these artefacts: retyping (compilation), simulation (execution), modification (modification), and sanitization (initialization and merging).

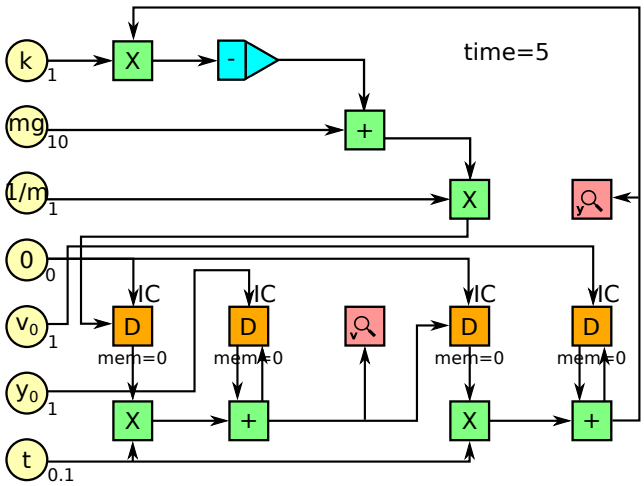
The **Retype** operation is the equivalent of the *compile* operation. Similar to compilation, it creates a semantically equivalent copy of a model, while retyping it to a runtime model. It does not necessarily have to be a trivial retyping, as potentially the design and partial runtime model have a slightly different structure (*e.g.*, flattening hierarchy). Retyping is thus also responsible for making this translation. As explained before, the partial runtime models for both the FSA and DTCBD formalism do not contain additional information. The retyping operation is therefore trivial in this case. For CTCBDs, the retyping actually casts the model to a formalism for which semantics exists. This operation involves discretization (one CTCBD element is mapped to multiple DTCBD elements) and optimization (multiple CTCBD ele-



(a) The full runtime model of the example FSA, during execution.



(b) The full runtime model of the example DTCBD, during execution.



(c) The full runtime model of the example CTCBD, during execution.

Fig. 12: The full runtime models of the examples.

ments are mapped to one DTCBD element). After this discretization, however, the case becomes identical to DTCBDs for the remainder of the live modelling process. In all cases, traceability links are created between the various elements to help in future operations. For example, in the FSA, the design state is linked to the equivalent partial runtime state, such that on subsequent operations, it is known that this state has already been converted before, and therefore does not need to be recreated again.

The **Simulation** operation is the equivalent of the *execution* operation. Simulation computes the next state of the full runtime model and updates it in-place. For the FSA formalism, the next state of the model is computed by processing an event from the environment, and executing an enabled transition by changing the current state and (optionally) raising output events to the environment. For the DTCBD and CTCBD formalism, there is no external input or output. The next state of the model is computed by, for each block, computing the output signal value based on its input values. This requires detecting loops and solving them if they represent a set of linear equations. For delay blocks, the output value is equal to its value in memory (or the initial condition at the first iteration when the memory value has not been set yet). The memory value is overwritten by the current input value of the delay block. At the end of computing the next value of all blocks' output signal values, the iteration counter is incremented. As we are operating on models, and not on generated code, we do not need to consider the technical aspects of replacing executing code: the model is updated in-place and the simulation algorithm picks up these changes in the next step. Note, however, that the simulation algorithm does *not* take care of initialization, as is usually the case. Indeed, normally the first step of simulation is to initialize variables, which is now unnecessary: all information is stored and read out from the model itself. Some parts of the simulation algorithm still need to be done, which are not related to initialization of the model, but initialization of the simulation algorithm, such as topological sorting for DTCBDs.

The **Modification** operation is the equivalent of the *modification* operation in programming. Similar to modification in the programming domain, users can only modify the design model. Since all other artefacts are automatically generated, the design model is the only artefact they are familiar with. While the user never edits the partial or full runtime models directly, the design model can be freely modified. As usual, Create-Read-Update-Delete (CRUD) operations are supported on the model. This boils down to Creating new elements and attributes, Updating the values of attributes, and Deleting elements and attributes. Note that reading does not modify the model, and is therefore ignored.

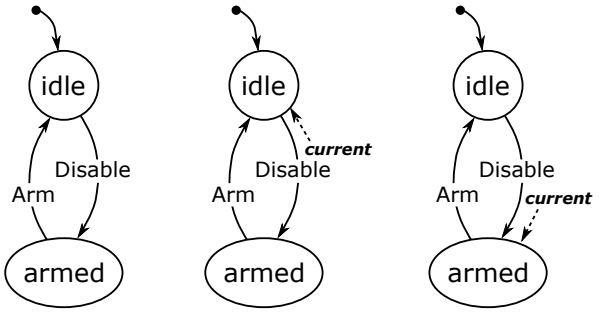
To highlight the various types of changes, each formalism has a different type of change. For the FSA formalism, users can change the triggers on transitions, remove transitions, create new states, and so on. A modified FSA design model is shown in Figure 13a, where the *detected* state is removed (Delete). For the DTCBD formalism, users can instantiate new blocks, delete existing blocks, add or remove dependencies, and so on. A modified DTCBD design model is shown in Figure 14a, where the value of $y(t)$ is multiplied by two, thereby changing the algebraic loop (Create). For the CTCBD formalism, users can instantiate new blocks (including the integrator and derivator), delete existing blocks, add or remove dependencies, and so on. A modified CTCBD design model is shown in Figure 15, where the gravitational constant is altered (Update). For all formalisms, the design models must

conform after the modifications. Note that different types of operations were applied for each formalism: removing a structural element in FSAs, creating several structural elements in DTCBDs, and changing a parameter in CTCBDs. This highlights the various types of operations that can be done on the design model, all of which are reflected in the running simulation.

The **Sanitization** operation is the equivalent of the *merge* operation. While it is indeed a merge operation, it was renamed to sanitization to prevent confusion with the existing term model merging [4]. The operation creates a full runtime model from a (new) partial runtime model and an (old) full runtime model. As the sanitization is domain-specific, it is difficult to make general claims about this operation: it is whatever the language engineer wants it to be. Nonetheless, the sanitization function can be sure that both input models will conform to their metamodel (which the language engineer can define), and must ensure its output conforms to the full runtime metamodel. Sanitization includes initialization (where the runtime state is empty) and the live modelling "merge", where the runtime state is taken into account. As discussed previously, sanitization is fundamental to live modelling support, and as such, it is discussed in detail next. As was the case with the retyping operation, sanitization makes use of traceability links, linking elements from the partial runtime to the full runtime. Traceability links are used to correctly migrate the state of the full runtime model to the right elements in the partial runtime model. For example, in the FSA, a state in the partial runtime model without any traceability link is considered to be a new element, while a state in the full runtime model without a traceability link is considered to be removed, possibly triggering a problematic situation when this was the current state of the simulation.

4.2.3 Sanitization

The sanitization operation is largely dependent on the types of changes to be merged (*i.e.*, breaking or non-breaking), but remains a formalism-specific operation. Therefore, a manually defined version needs to be created for each new formalism. Nonetheless, our decomposition has shown that this is the only operation that needs to be added, in order to provide live modelling for that formalism. Depending on how the sanitization operation is implemented, any of the three types of live modelling (*i.e.*, none, recorded event, or real-time) can be implemented. We leave open the medium in which this operation is expressed (*e.g.*, procedurally using code or declaratively using model transformations). The presented code snippets therefore do not restrict sanitization to a procedural approach. In this subsection, we present the sanitization operations for both types of state, using our running example: the FSA, DTCBD, and CTCBD formalisms. For all three, we present real-time live modelling. Note that, similar to live



(a) Modified design model. (b) Automatic resolution. (c) Manual resolution to "armed" state.

Fig. 13: Sanitization in FSAs.

programming, sanitization can only happen when the state is consistent (i.e., inbetween two execution steps).

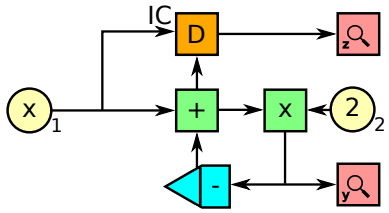
We have opted for an operational approach to define the sanitization operation, mostly for didactic reasons. Another approach would be denotational, for example through constraint solving. In that case, the models, metamodels, and all semantics would have to be encoded in a constraint system.

Breaking Changes When breaking changes are possible, the runtime model might have to be made conforming to its metamodel again. For example, when users remove the current state in the design model, the equivalent state in the runtime model also has to be removed, thereby violating the constraint that the runtime model has exactly one current state. In that case, a new state of the updated running system must be defined. Changes to any other aspect of the design model are irrelevant to the running system, and are just taken over.

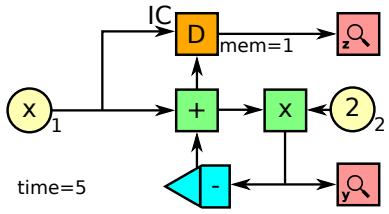
Algorithm 3 The FSA sanitize operation.

```
function SANITIZEFSA(M_P^new, M_F^old)
    if isInitialized() then
        currState ← getCurrentState(M_F^old)
        if not currState ∈ M_P^new then
            if automaticResolution then
                currState ← getInitialState(M_P^new)
            else
                if disallowChange then
                    raise Exception
                else
                    currState ← userChoice(M_P^new)
                end if
            end if
        end if
    else
        currState ← initializeState(M_P^new)
    end if
end function
```

Resolving this breaking change is the core task of the sanitization operation. There are several options: reset the current state to the initial state or pick the last known state



(a) New design model.



(b) Sanitized runtime model.

Fig. 14: Sanitization in DTCBDs.

(automated, so resolvable), prompt the user for a new state (manual, so non-resolvable), disallow the change completely (disallow breaking changes), and so on. For the new design model in Figure 13a and the old full runtime model in Figure 12a, the first two options are presented. Figure 13b shows automatic resolution where, in this case, the system chooses the initial state (the "idle" state) as the new current state. Figure 13c shows manual resolution, where the user chooses the "armed" state as the new current state. Figure 3 shows the pseudocode of a sanitize operation for FSAs, allowing for three different sanitization options.

Changes resulting in an undefined current state could also be explicitly disallowed. We did not pursue the direction of disallowing design model changes, as we explicitly want all modifications to be possible.

Non-Breaking Changes. For non-breaking changes, any change the user makes always reflects on a conforming runtime model. In contrast to breaking changes, where resolution is required, non-breaking changes don't require significant changes to the runtime model.

In our example DTCBD formalism, only operations on the integrator, derivator, and delay blocks have any influence. Since each block and connection has its own *signal* and *memory*, removing a block or connection only affects that specific signal. In further simulation steps, however, the change will of course have its effects on other elements as well, as it propagates through the system. It is possible, however, to add new parts to the state (*i.e.*, add new blocks or connections) or remove parts of the state.

When sanitizing, we take the structure from the partial runtime model, which we augment with the runtime data from the full runtime model. In the case of DTCBDs, the runtime information consists of (1) the current simulation time; and (2) the memory of delay blocks, derivators, and integrators. Blocks that were not present in the full runtime model are initialized as usual, since they are new. Blocks that were present, however, have their state copied from the full runtime model. The pseudocode of the sanitize operation for DTCBDs is shown in Algorithm 4.

Algorithm 4 The DTCBD sanitize operation.

```
function SANITIZECBD(M_P^new, M_F^old)
    for all block ∈ M_P^new do
        if block ∈ M_F^old then
            oldSignal ← getSignal(M_F^old, block)
            setSignal(M_P^new, block, oldSignal)
        else
            initializeSignal(M_P^new, block)
        end if
    end for
    if isInitialized() then
        iterations ← getNumberOfIterations(M_F^old)
        setNumberOfIterations(M_P^new, iterations)
    else
        initializeNumberOfIterations(M_P^new)
    end if
end function
```

An example of sanitization is shown in Figure 14. In this figure, we see the new design model in Figure 14a, and the resulting full runtime model in Figure 14b. The full runtime model consists of the structure of the partial runtime model, combined with the values of the old full runtime model. In this case, the value of the *t* variable (representing the current iteration of the simulation), as well as the memory value of the delay block, are copied.

Denotational Semantics. For denotational semantics, the sanitization is done at the level of the target formalism, and will therefore be any of the previously mentioned approaches. Sanitization might require traceability information to be present. This information links the various models to be merged together, as indeed the source and target partial runtime models can vary significantly. Using traceability links, elements in different formalisms can be connected to their equivalent counterparts. Some elements, such as the integrator in CTCBDs, will have traceability links to multiple elements in the DTCBD partial runtime model, as it was expanded (1-to-n mapping). Other elements, such as a constant block in CTCBDs, might have traceability links to a shared element in the DTCBD partial runtime model, as it was partially optimized away (n-to-1 mapping).

The sanitization process is completely identical to that of operational semantics in terms of traceability information: information is stored during retyping and sanitization, and is subsequently used in the next sanitization phase to identify equivalent elements. The only difference is that there is no longer a 1-to-1 mapping, but an n-to-n mapping. Nonetheless,

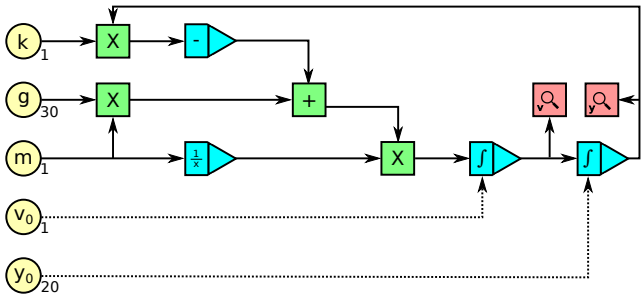


Fig. 15: New design model for CTCBDs.

during all phases of live modelling, traceability links are still created. Using this information, it is still possible to find out which design element(s) was the source of the current element in the full runtime model. As for each element in the full runtime model the design element is known, it is possible to find out which elements are identical and should have their state copied.

No new sanitize operation is presented, as the DTCBD sanitization operation is reused.

4.3 Live Modelling Process

An overview of the approach, for each case, is shown in Figure 16 for FSAs, in Figure 17 for DTCBDs, and in Figure 18 for CTCBDs.

More generally, Figure 19 shows an FTG+PM [23] model describing both the different formalisms and processes of live modelling for any formalism. The left side shows the Formalism Transformation Graph (FTG), describing the different formalisms and the transformations between them. The right side shows the Process Model (PM), describing the sequence of operations done by the user and the data dependencies. It includes the artefacts, how they are related, and the process describing the (automatic or manual) operations. The sanitize operation has a dual colour: it is mostly automatic, though it can be manual for non-resolvable breaking changes, where the user is prompted. In the PM, simulation and modification run concurrently: modifications can be made throughout simulation. This is typical for live modelling, in contrast to the mostly linear development process of a single model in ordinary modelling.

4.4 Relation to Multi-Paradigm Modelling

Our approach can be considered a Multi-Paradigm Modelling (MPM) approach to live modelling for several reasons.

On the one hand, this approach builds on MPM, as it requires all techniques that are present in MPM: language engineering (e.g., for domain-specific modelling), activities (e.g., for model transformations), and processes (e.g., for enactment). Language engineering is required for the various formalisms that are used by the approach: design metamodel, partial runtime metamodel, and the full runtime metamodel. All these formalisms must be created within the tool and should have support for maintaining them. Activities are required to relate the various formalisms and models together, thereby automatically applying the approach. Activities can be implemented in different ways, such as through declarative model transformations or a procedural action language, and are executed to translate between the various models. Processes are required to structure the approach, thereby preventing it from being ad-hoc as the majority of other approaches to liveness. With support for enactment, it even becomes possible to automatically perform the complete live modelling approach. In conclusion, all relevant aspects of the approach are modelled explicitly, as proposed by MPM.

On the other hand, this approach is desirable in an MPM context, as MPM requires the use of the most appropriate formalism(s) for a problem. The most appropriate formalism, however, is likely to be domain-specific and have a rather limited application domain. As such, the number of users of these formalisms is small, making it hard to justify the effort ordinarily required to make formalisms live. With the proposed generic approach, formalisms can more easily be made live with the addition of a “sanitize” operation, significantly lowering the threshold to live modelling and increasing the usability of the formalism. Increasing the usability of a formalism naturally makes the formalism more appropriate for its use, thereby strengthening the MPM approach.

5 Implementation

To assess the feasibility of our approach, we implemented live modelling for the three running examples. Our prototype consists of a single visual modelling and simulation front-end, in which multiple formalisms can be loaded, including FSAs, DTCBDs, and CTCBDs. This front-end is unaware of live modelling. All operations are defined in the Modelverse [47], our Multi-Paradigm Modelling (MPM) tool. The Modelverse implements all aspects of MPM [48], making it possible to use all aspects of language engineering, model transformations, and process modelling, as required by our approach.

In our prototype tool, users start the live modelling process relevant to the formalisms they want to use. The process can be parameterized with an input model, which is the initial model. If no input model is provided, users start from an empty model. Independent of the initial model, simulation is always started anew, as only the design models are stored. Enactment completely resembles the usual modelling interface, but instead of only having a modelling window, a simulation window is now also present. This simulation

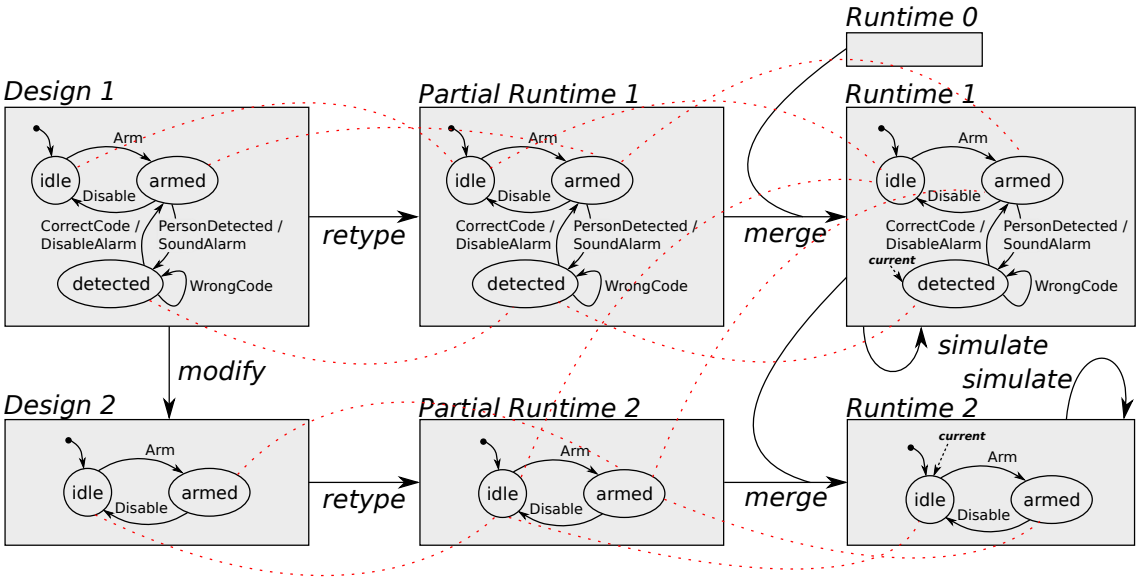


Fig. 16: Overview of our approach applied to an FSA mode, including traceability links.

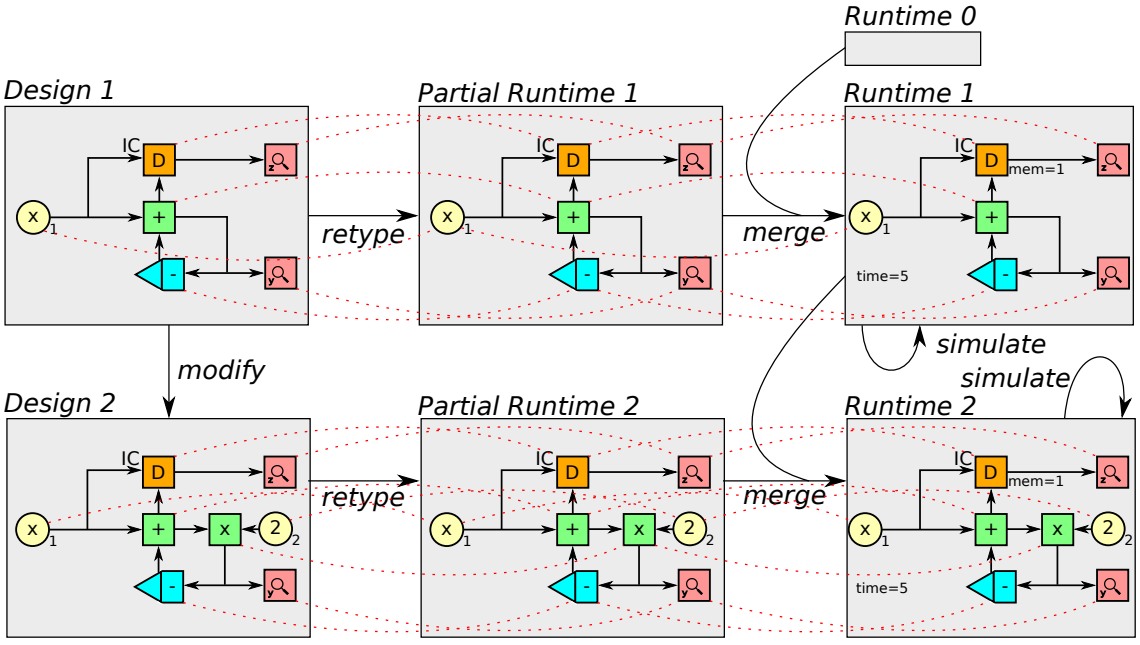


Fig. 17: Overview of our approach applied to a DTCBD model, including traceability links.

window is merely an external program that visualizes the simulation results obtained.

Even during modelling, simulation is progressing, and users will see that the simulation window is updated in real-time. Changes made by the user are not immediately committed to the actual design model, as users might want to group a set of operations together into a transaction. As soon as users are satisfied with the design model, and wish to propagate the changes to the running simulation, they commit the design model. In our prototype implementation, committing can be done by closing the modelling window. When the window is closed, the manual "edit" activity is finished, and the process enactment continues by stopping the current execution and performing the required translations. Once these are completed, simulation is resumed and a new modelling window is opened with the current version of the design model. Users will immediately see that their simulation is resumed, but now taking into account the new model.

For all the three examples presented below, the exact same tool is used, with the exact same (parameterized) FTG+PM model. Apart from the usual operations that have to be implemented for any formalism (i.e., runtime metamodels, opera-

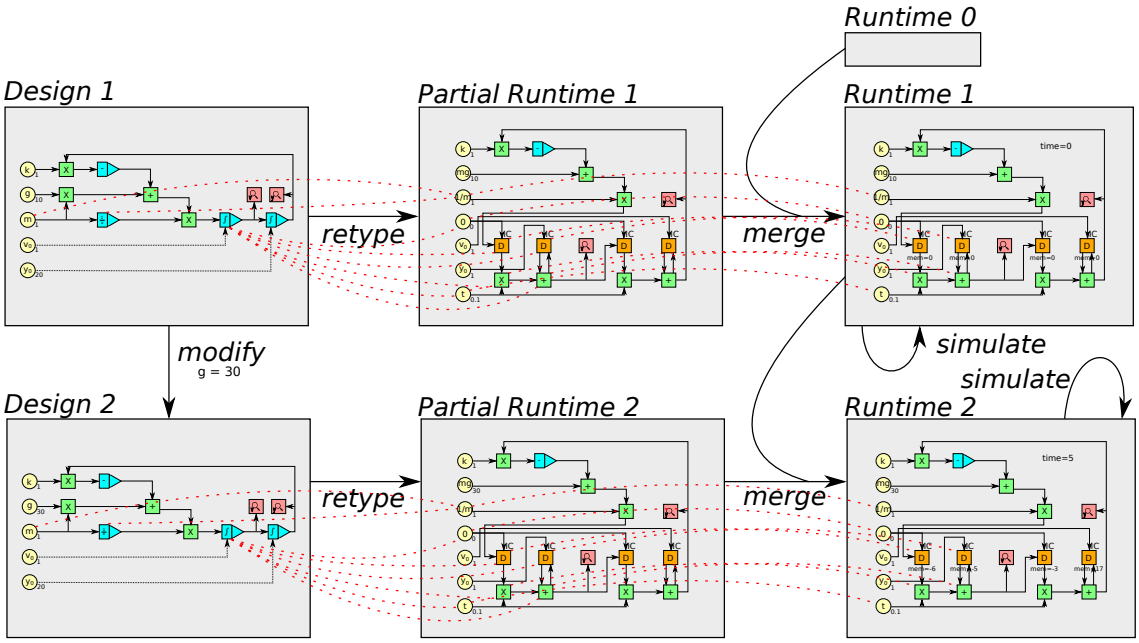


Fig. 18: Overview of our approach applied to a CTCBD model. Only some interesting traceability links are shown.

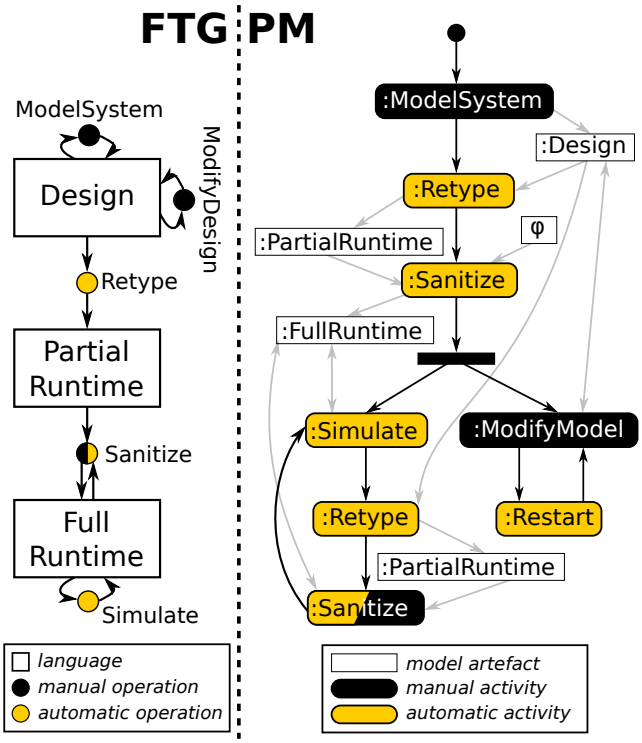


Fig. 19: Overview of our approach, as an FTG+PM model.

tional semantics, denotational semantics), only the sanitize operation is new, and had to be defined for each formalism individually. As for the visual interfaces, these are untouched when implementing live modelling, as everything is based on process enactment.

5.1 Finite State Automata

The implementation of our FSA live modelling environment is shown in Figure 20. To the left, the modelling window is shown, which contains a visual representation of the design model. To the right, the simulation window is shown, which is continuously updated with results from the running simulation. The trace shows the current state throughout time. Although FSAs are untimed, input events can be raised by the user through the simulation interface. The state of the system is constant in between such events; the time plotted on the x-axis is wall-clock time. The FSA model itself is oblivious of the current time.

During exection, the current state ("idle") is removed and the new initial state is set to "armed". Upon committing this change, the model and trace is updated as shown in Figure 21. It is shown that, upon making that change, sanitization sets the new current state to the new initial state, which is "armed" in this case. Note that there will always be a single initial state, as this is part of the constraints imposed by the metamodel. The history of the simulation is left as-is, since the history is not rewritten with real-time live modelling. Nonetheless, the current state has no effect on the result of sanitization.

5.2 Discrete Time Causal Block Diagrams

The implementation of our DTCBD live modelling environment is shown in Figure 22. It is similar to the FSA live modelling environment, as they reuse a lot from the Modelverse and our generic approach. Actually, the only difference

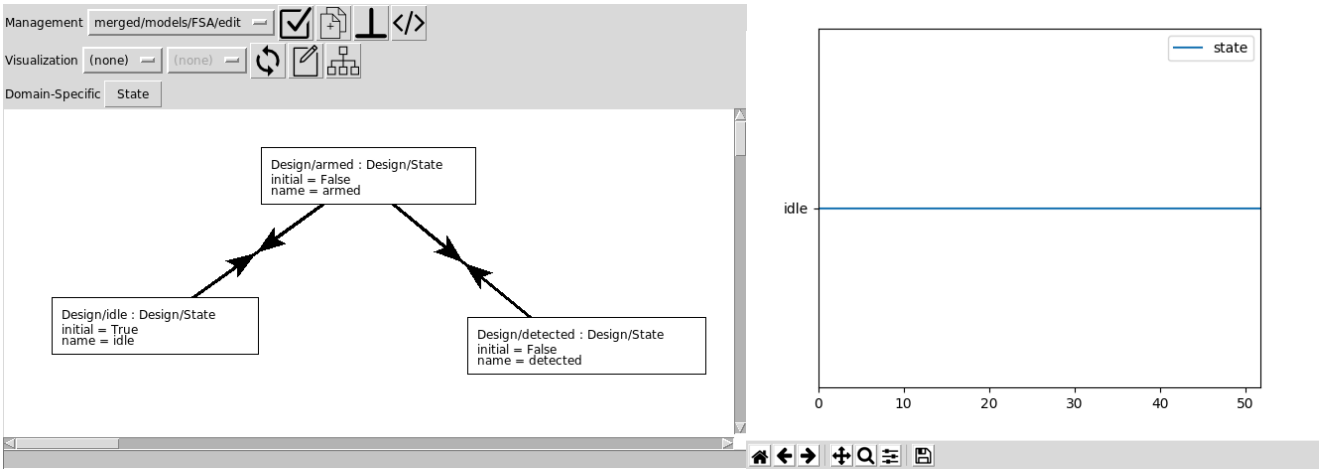


Fig. 20: Live modelling for FSAs, before change.

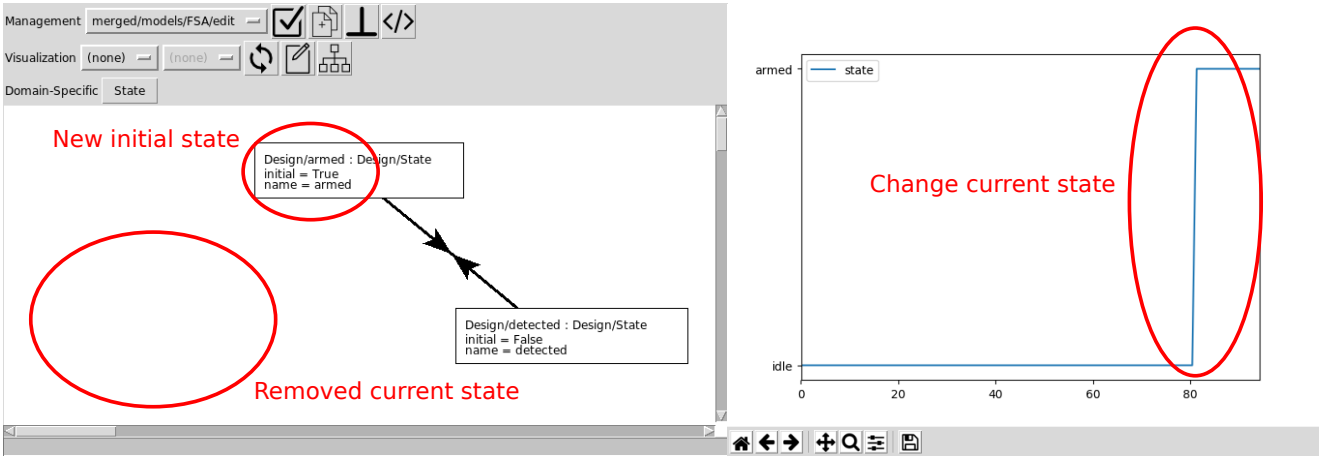


Fig. 21: Live modelling for FSAs, after removing current state and setting new initial state.

related to live modelling is the sanitize operation. Of course, the formalisms also differ, just like the simulation viewer, though these are all independent of live modelling, and would be required anyway, even without live modelling. In the simulation view, probed signals are plotted. It is possible for signals to appear or disappear throughout simulation, when a probe block is added or removed during simulation. This is a design consideration of the simulation viewer if it wants to support live modelling.

During execution, the algebraic loop is resolved and sets both y and z to $\frac{1}{2}$. After some time, the algebraic loop in the DTCBD model is extended with an additional multiplication block and constant 2. The value for z now becomes the output of the addition block, while the value for y becomes the result of the multiplication block. After all elements are connected and changes are committed, the trace is updated, as shown in Figure 23. Again, the algebraic loop is solved transparently to the user, resulting in a y value of $\frac{2}{3}$ and a z value of $\frac{1}{3}$

5.3 Continuous Time Causal Block Diagrams

The implementation of our CTCBD live modelling environment is shown in Figure 24. It is identical to the DTCBD live modelling environment, but now we have access to the derivator and integrator blocks. To the user, it is indistinguishable whether this live modelling functionality was provided by using an operational or denotational semantics approach. Similarly, the simulation viewer from DTCBDs is reused.

Up to time 60, the simulation executes the model shown in Figure 24, showing the results on the trace in Figure 25. We notice the harmonic oscillator behaviour that is expected of such a system. At time 60, however, the CTCBD model is altered by changing the value of constant g from 10 to 30, effectively being a sudden increase in gravitational force. This has an immediate effect on the simulation trace, as shown in Figure 25 after time 60: instead of having a decreasing velocity, the velocity starts increasing again. Results stay con-

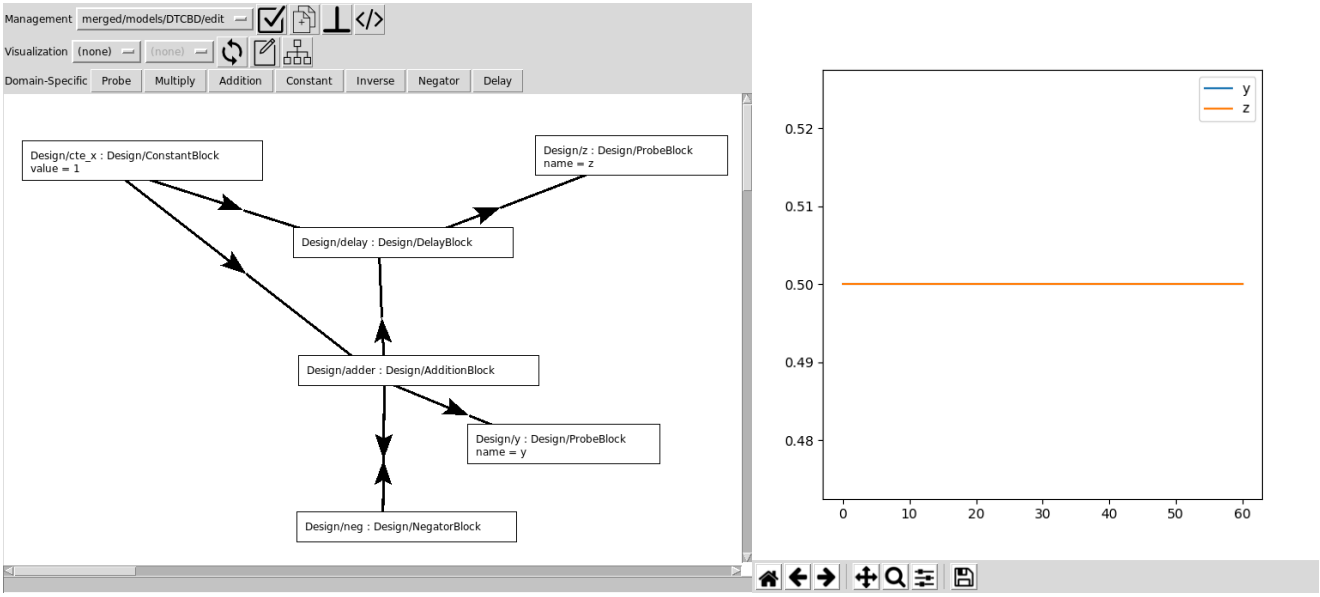


Fig. 22: Live modelling for DTCBDs, before change.

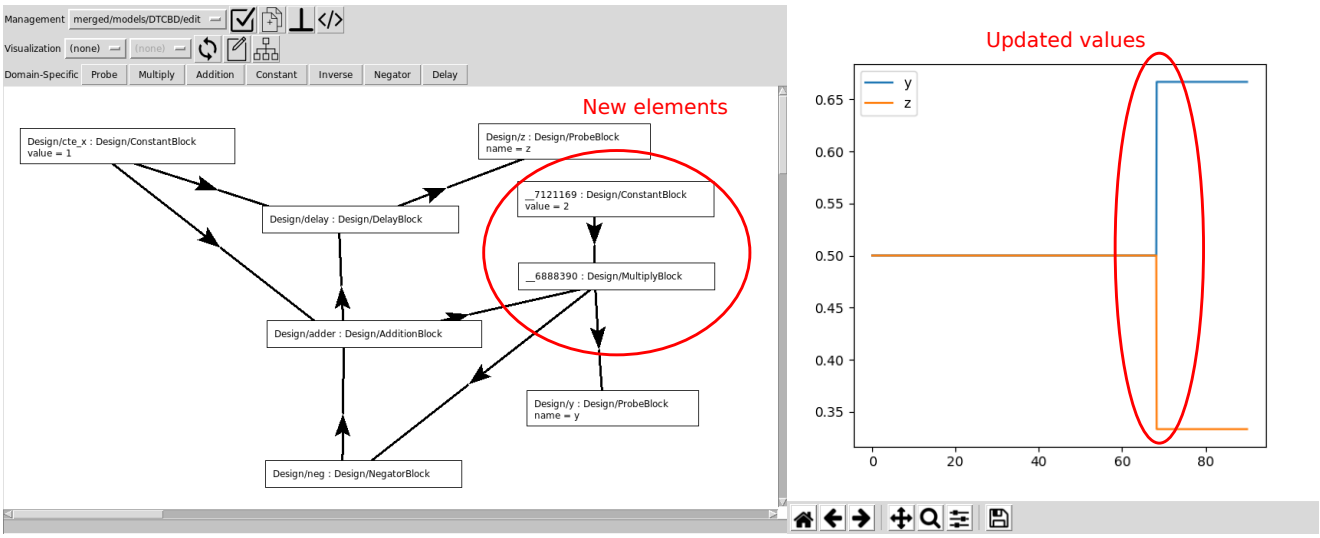


Fig. 23: Live modelling for DTCBDs, after adding and connecting the multiplication block.

tinuous, though a difference in behaviour is clearly observed at the point in simulation where the change was made.

5.4 Discussion

Given that our generic tool could be used for three different domain-specific formalisms, while all using the same (parameterized) FTG+PM model, we believe our approach to be applicable to a wide variety of modelling formalisms. Indeed, our structured approach required no modification for these three types of semantics, making us believe that it can be applied for other formalisms as well. We can therefore assume that our approach provides structure to the currently ad-hoc process of making a formalism live.

Sanitization was the only activity that was further required; its logic was described in the previous sections. The goal of the sanitization function is conceptually clear: combine the currently executing model (with state information) with an uninitialized runtime model. In the limit, this sanitization function can be seen as an advanced initialization function, which can take an existing simulation model as input. We can therefore assume that our approach can make existing formalisms live with little additional work for the language engineer.

As presented in this paper, the sanitization operations are relatively small in size and easy to understand, but of course these are still relatively simple example formalisms. In our examples, the sanitization operation is actually rather

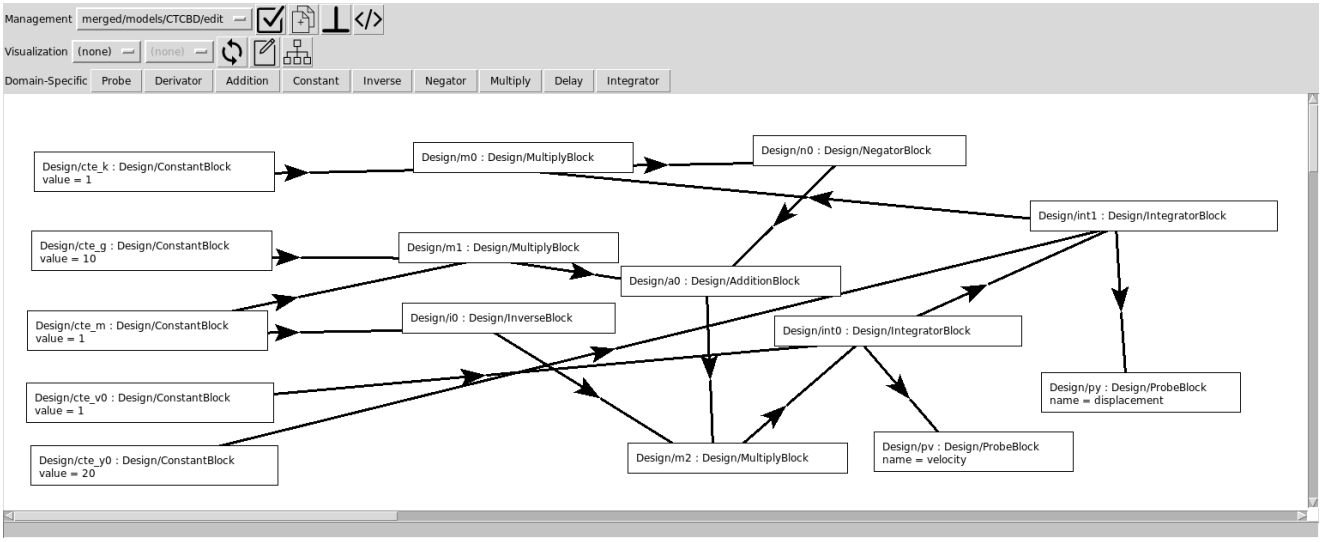


Fig. 24: Implementation of live modelling for CTCBDs.

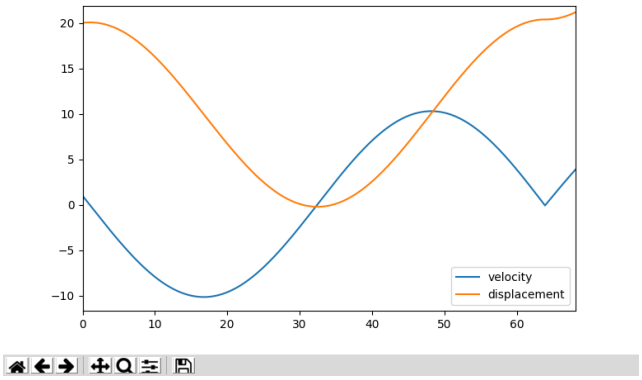


Fig. 25: Simulation trace, where the constant “g” is changed at around time 63.

efficient as well, given its low time complexity. Note that performance of the simulation is not impeded, as the simulation activity in itself is not modified at all: it only pauses upon changes, where the model has to be reloaded. This is particularly obvious with CTCBDs, where the translation is only done once, and the execution stays at the level of DTCBDs. As such, there is no negative impact on normal simulation performance.

6 Related Work

Our contribution provides a formalism-neutral overview of liveness, thus enabling liveness for domain-specific formalisms. Two categories of related work exist: live programming and (live) executable modelling.

In the live programming domain, the concept of liveness is well-studied. One of the most important distinctions between different approaches is how they handle time: a distinction is made between *real-time* and *recorded event* [26]. In real-time mode, the past is left unaltered, and only future executions of the code are influenced. This is often termed *fix and continue*, as implemented by Lisp [38], Smalltalk [14], Erlang [1], and SELF [44]. In recorded event, all past input events are recorded and replayed, resulting in a completely new history. This is implemented in languages such as ElmScript [9] and YinYang [26]. We only implement the real-time live modelling approach, as recorded live modelling has been shown not to be ideal for simulation [26]. Nonetheless, further investigation into recorded event live modelling might be interesting for other types of formalisms.

A lot of work is spent towards making live programming usable. This requires research as to which representation is most usable, such as textual or graphical formalisms [25, 35, 15, 11]. Therefore several kinds of formalisms have been made live: graphical formalisms such as VIVA [43] and Flogo [16], textual formalisms such as ElmScript [9] and Smalltalk [14], and hybrid formalisms such as Subtext [11]. Our approach does not commit itself to textual, graphical, or hybrid formalisms. It is implemented on the abstract syntax of models, and does not require a specific visualization. If required, our live simulator can be coupled to multiple interfaces, possibly with different representations (e.g., textual, graphical).

Another important usability aspect of live programming is the need for immediate feedback to the user [43], resulting in the need for effective visualization and tight latency constraints [25, 40]. Latency is considered harmful when it becomes too large, with the threshold being defined some-

where between $50ms$ [26] and $500ms$ [25]. For this reason, a lot of work has focused on optimizing specific aspects, such as incremental compilation [26] and code hotswapping [12]. Our framework focuses exclusively on the functional requirements of live modelling, without considering performance, visualization, and so on. While these concerns are certainly important, we consider them as future work.

Many challenges related to live modelling are tackled only for specific cases or specific formalisms. An example issue is the question how the state needs to be retained [12, 41], and what needs to be recomputed [6]. Making an existing programming language live is often done through ad-hoc modifications, often turning liveness into a black art [5]. With our approach, we provide an overview of the steps required to make a formalism live. And while not fully automated, since some domain information remains necessary, the approach becomes structured and easier for language engineers to understand and implement.

In the modelling domain, the focus has primarily been on the theoretical foundations of (meta-)modelling [20] and how (domain-specific) modelling can help developers [18]. Nowadays, focus starts shifting to model execution [27] and debugging [24]. And whereas model debugging is often formalism-specific, such as for Causal Block Diagrams [50] and Parallel DEVS [46], recently new approaches have been developed that try to (partially) automate the addition of debugging to formalisms [45]. Advanced tracing facilities for domain-specific formalisms have been developed [3], which enable omniscient, or backwards-in-time debugging [2]. Closer to our approach is [42], in which the author explores how formalisms can be made live with "semantic deltas". The system is capable of translating source program modifications (so-called deltas) to operations on the running code. While the paper presents a prototype demonstrating the approach, it does not present a structured way to add live modelling to formalisms. Similarly, another approach is based on textual differences [37], where existing textual difference algorithms are leveraged to update the executing model. While that approach is also relatively generic, it focuses exclusively on textual formalisms, and is only evaluated in the context of one kind of finite state automaton. Since live modelling is rarely implemented, or at best in an ad-hoc way, we contribute by providing a general framework for merging liveness into existing modelling formalisms, paired with an implementation for three example formalisms.

Similar to reflection and code hotswapping, the modelling community is starting to acknowledge the existence of models at runtime. These models, however, are mostly used for self-managing systems [36, 29], and do not directly apply to live modelling. Specifically, models at runtime make the changes internally, as a part of pre-defined, correct behaviour. Live modelling, on the other hand, makes changes due to external operations, knowing that some part of the model may be incorrect. Additionally, models at runtime techniques are used to express dynamicly changing systems, whereas live modelling is used for modifiable systems (e.g., for debugging or education). Due to this mismatch in application domain, their requirements severely differ. For example, models at runtime do not need to cope with changes at the design model, but applies changes on the full runtime model, rendering sanitization unnecessary.

Finally, model evolution [13], and in particular language evolution, has similar challenges to code hotswapping. When swapping code, but retaining the state, the old state might not be understandable for the new code operating on it [12]. Similarly, language evolution tries to tackle the problem of existing models not being updated after a language change. Sanitization, as part of model co-evolution [28], tackles such changes semi-automatically. Our sanitization approach is similar, as we also need to adapt a model under execution to an evolved design model.

7 Conclusion

In this paper, we have argued in favour of live modelling: live programming transposed to modelling. Our framework is based on a deconstructed process of live programming, which was reconstructed for modelling. The reconstruction process transposes operations on programming artefacts (*e.g.*, compilation, initialization, and execution) to equivalent operations on model artefacts. As we do not require specific formalism features, our framework is applicable to many formalisms, including general-purpose programming languages and domain-specific modelling formalisms. Using our approach, adding liveness to (domain-specific) modelling formalisms becomes more structured and reproducible, though still necessarily manual. The effort of making a formalism live is completely shifted to defining a sanitize operation, with all previously defined operations remaining untouched. Domain-specific modelling formalisms profit from this contribution, as it becomes easier to add advanced concepts to formalisms with a small user base. Due to the sheer number of domain-specific modelling formalisms that we envision in the near future, a structured approach will certainly help to make them live.

As an example of our approach, we have applied this framework to three formalisms: Finite State Automata (operational semantics with breaking changes), Discrete Time Causal Block Diagrams (operational semantics with non-breaking changes), and Continuous Time Causal Block Diagrams (denotational semantics). All these modelling formalisms have distinct characteristics, demonstrating that our approach is widely applicable. For each, a new sanitize activity was defined, while reusing all other operations and processes, which was sufficient to support live modelling.

Further research is required to improve the performance of our approach, as this was not an objective of this work. There are two time-consuming operations in our approach: the "retype" operation, which has to operate on the complete model, and the "sanitize" operation. Incremental approaches to retyping might serve in the modelling community to speed things up, particularly in combination with incremental model transformations. The performance of the sanitize operation depends significantly on the formalism, as this might range from trivial (e.g., reset to the initial state) to complex (e.g., use constraint satisfaction to find the new state).

The combination of live modelling with other debugging operations is another interesting direction, as these features might interact with each other. For example, omniscient debugging allows users to step back in time to inspect the state *before* the bug manifests itself. In combination with live modelling, this would require model changes to be taken into account as well, thereby stepping not only through states, but also through model structures.

Denotational semantics could possibly also be used to map domain-specific formalisms onto a live programming language, thereby making the model live, while relying on existing live programming techniques.

Acknowledgements This work was partly funded by PhD fellowships from the Research Foundation - Flanders (FWO) and Agency for Innovation by Science and Technology in Flanders (IWT). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

1. Armstrong, J.: The development of Erlang. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97, pp. 196–203. ACM, New York, NY, USA (1997). DOI 10.1145/258948.258967
2. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, pp. 137–148. ACM, New York, NY, USA (2015)
3. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In: 11th European Conference on Modelling Foundations and Applications (ECMFA). L'Aquila, Italy (2015)
4. Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh, M.: A manifesto for model merging. In: Proceedings of the 2006 International Workshop on Global Integrated Model Management, GaMMa '06, pp. 5–12. ACM, New York, NY, USA (2006). DOI 10.1145/1138304.1138307
5. Burckhardt, S., Fähndrich, M., Kato, J.: It's alive! continuous feedback in UI programming. In: Proceedings of PLDI '13, pp. 95–104 (2013)
6. Burnett, M.M., Atwood Jr., J.W., Welch, Z.T.: Implementing level 4 liveness in declarative visual programming languages. In: Proceedings of Visual Languages '98, pp. 126–133 (1998)
7. Cellier, F.E.: Continuous System Modeling. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1991)
8. Chiş, A., Denker, M., Gîrba, T., Nierstrasz, O.: Practical domain-specific debuggers using the Moldable Debugger framework. Computer Languages, Systems & Structures **44**(A), 89–113 (2015)
9. Czaplicki, E.: Elm: Concurrent FRP for functional GUIs. `https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf` (2012)
10. Déva, G., Kovács, G.F., Ancsin, A.: Textual, executable, translatable UML. In: Proceedings of the Workshop on OCL and Textual Modeling Applications and Case Studies, pp. 3–12 (2014)
11. Edwards, J.: Subtext: Uncovering the simplicity of programming. In: Proceedings of OOPSLA '05, pp. 505–518 (2005)
12. Fabry, R.S.: How to design a system in which modules can be changed on the fly. In: Proceedings of ICSE '76, pp. 470–476 (1976)
13. Favre, J.M.: Languages evolve too! changing the software time scale. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution, IWPSE '05, pp. 33–44. IEEE Computer Society, Washington, DC, USA (2005). DOI 10.1109/IWPSE.2005.22
14. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
15. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Text-based modeling. In: Proceedings of the 4th International Workshop on Software Language Engineering (2007)
16. Hancock, C.M.: Real-time programming and the big ideas of computational literacy. Ph.D. thesis, Massachusetts Institute of Technology (2003)
17. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
18. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley (2008)
19. Kuhn, A., Murphy, G.C., Thompson, C.A.: An exploratory study of forces and frictions affecting large-scale model-driven development. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems, MODELS'12, pp. 352–367. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-33666-9_23
20. Kühne, T.: Matters of (meta-)modeling. Software and System Modeling **5**, 369–385 (2006)
21. Lieberman, H., Fry, C.: Bridging the gulf between code and behavior in programming. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 480–486 (1995)
22. Lindeman, R.T., Kats, L.C.L., Visser, E.: Declaratively defining domain-specific language debuggers. In: Proceedings of the 10th International Conference on Generative Programming and Component Engineering, pp. 127–136 (2011)
23. Lucio, L., Mustafiz, S., Denil, J., Vangheluwe, H., Jukss, M.: FTG+PM: An Integrated Framework for Investigating Model Transformation Chains. In: SDL 2013: Model-Driven Dependability Engineering, *Lecture Notes in Computer Science*, vol. 7916, pp. 182–202. Springer (2013). DOI 10.1007/978-3-642-38911-5_11
24. Mannadiar, R., Vangheluwe, H.: Debugging in domain-specific modelling. In: B. Malloy, S. Staab, M. Brand (eds.) Software Language Engineering, *Lecture Notes in Computer Science*, vol. 6563, pp. 276–285. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-19440-5_17
25. McDirmid, S.: Living it up with a live programming language. In: Proceedings of OOPSLA '07, pp. 623–638 (2007)
26. McDirmid, S.: Usable live programming. In: Proceedings of Onward! 2013, pp. 53–61 (2013)
27. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley Professional (2002)
28. Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Sci. Comput. Program. **76**(12), 1223–1246 (2011). DOI 10.1016/j.scico.2011.01.002

29. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer **42**(10), 44–51 (2009). DOI 10.1109/MC.2009.327
30. Mosterman, P.J., Vangheluwe, H.: Computer automated multiparadigm modeling: An introduction. SIMULATION **80**(9), 433–450 (2004). DOI 10.1177/0037549704050532
31. Murata, T.: Petri nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4), 541–580 (1989). DOI 10.1109/5.24143
32. National Science Foundation: Cyber-Physical Systems (CPS). `https://www.nsf.gov/pubs/2016/nsf16549/nsf16549.pdf` (2016). Document number: nsf16549
33. Oakes, B.: Optimizing simulink models. Tech. Rep. CS-TR-2014.5, McGill University (2014)
34. Pavletic, D., Voelter, M., Raza, S.A., Kolb, B., Kehrer, T.: Extensible debugger framework for extensible languages. Lecture Notes in Computer Science **9111**, 33–49 (2015)
35. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Commun. ACM **38**(6), 33–44 (1995). DOI 10.1145/203241.203251
36. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Model-driven development of self-managing software systems. In: Proceedings of the Models at run.time workshop co-located with the ACM/IEEE 9th International Conference MODELS 2006 (2006)
37. van Rozen, R., van der Storm, T.: Towards live domain-specific languages: from text differencing to adapting models at run time. Software & Systems Modeling pp. 1–18 (2017)
38. Sandewall, E.: Programming in an interactive environment: The "lisp" experience. ACM Comput. Surv. **10**(1), 35–71 (1978). DOI 10.1145/356715.356719
39. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003). DOI 10.1109/MS.2003.1231150
40. Sorensen, A., Gardner, H.: Programming with time: cyber-physical programming with Impromptu. In: Proceedings of Onward! 2010, pp. 822–834 (2010)
41. Stewart, D., Chakravarty, M.M.: Dynamic applications from the ground up. In: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, pp. 27–38 (2005)
42. van der Storm, T.: Semantic deltas for live DSL environments. In: Proceedings of the 1st International Workshop on Live Programming, LIVE '13, pp. 35–38. IEEE Press, Piscataway, NJ, USA (2013)
43. Tanimoto, S.L.: VIVA: A visual language for image processing. Journal of Visual Languages and Computing **1**, 127–139 (1990)
44. Ungar, D., Smith, R.B.: Self: The power of simplicity. SIGPLAN Not. **22**(12), 227–242 (1987). DOI 10.1145/38807.38828
45. Van Mierlo, S.: Explicitly modelling model debugging environments. In: Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015, pp. 24–29 (2015)
46. Van Mierlo, S., Van Tendeloo, Y., Vangheluwe, H.: Debugging Parallel DEVS. SIMULATION **93**(4), 285–306 (2017). DOI 10.1177/0037549716658360
47. Van Tendeloo, Y.: Foundations of a multi-paradigm modelling tool. In: Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015 (2015)
48. Van Tendeloo, Y., Vangheluwe, H.: The Modelverse: a tool for multi-paradigm modelling and simulation. In: Proceedings of the 2017 Winter Simulation Conference, WSC 2017, pp. 944 – 955. IEEE (2017)
49. Vangheluwe, H., de Lara, J., Mosterman, P.J.: An introduction to Multi-Paradigm Modelling and Simulation. In: Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), pp. 9 – 20 (2002)
50. Vangheluwe, H., Riegelhaupt, D., Mustafiz, S., Denil, J., Van Mierlo, S.: Explicit modelling of a CBD experimentation environment. In: Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS, TMS/DEVS '14, part of the Spring Simulation Multi-Conference, pp. 379–386. Society for Computer Simulation International (2014)
51. Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. Software: Practice and Experience **38**(10), 1073–1103 (2008)
52. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2005)