Pragmatic Action Charts

Steven Smyth
Programming Systems Group
TU Dortmund University, Germany
0000-0003-2470-0880

***Abstract*—Pragmatic Action Charts are a minimalist Statecharts variant that can be implemented relatively easily as internal DSL in most mainstream languages. In this paper, they are implemented as internal DSL leveraging the object-oriented paradigm of the host language avoiding the introduction of yet another programming language. Internal DSLs can also benefit from appropriate instantaneous visualizations, which is usually attributed to specialized, graphical DSLs.**

Pragmatic Action Trees use the same minimalist approach to integrate behavior trees into an arbitrary host language. To achieve this, they are built on top of Pragmatic Action Charts. In fact, both notations can be mixed arbitrarily to allow behavior trees to be called if the chart is currently in a specific location and vice versa, the chart can transfer control to another location as soon as a behavior tree finishes its execution.

***Index Terms*—programming languages, domain-specific languages, statecharts, pragmatics, transient views, behavior trees**

I. Introduction

There are many attempts to combine State Machines (SMs) with General Purpose Languages (GPLs), such as the UML [1] or various synchronous languages, e. g., Esterel [2]. They are relatively easy to model both, in source code and graphically. Commonly, the developer starts with one form or the other to generate the desired artifacts. For example, UML diagrams are often used to generate source code stubs and vice versa, SyncCharts [3] can give a graphical Statecharts [4] representation to an Esterel program. Usually, these approaches employ their own DSL and Model of Computation (MoC). Despite their usefulness and decades long presence in various forms, they are arguably often more seen as a pattern that should be implemented from scratch when the need arises, rather than being a stable component in common mainstream programming languages. This is hardly surprising considering the sheer amount of different approaches and MoCs [5]. Besides the safety-critical domain, where the pressure and resources are large enough, also common projects, such as small size reactive systems, which execute periodically [6], behavior trees in robotics [7], or game loops [8], benefit from the SM pattern. I argue that the simplicity and therefore usability of the SM pattern can be increased, especially for more common projects.

I therefore present (i) the minimalist Pragmatic Action Charts (PACs) that can be easily realized in most modern programming language in Sec. II, (ii) the Pragmatic Action Trees (PATs) extention for behavior trees in Sec. III, and (iii) a demonstration of convenient tool integrations, namely transient views of internal DSLs and compatible external DSLs in Sec. IV. I discuss related work in Sec. V and conclude in Sec. VI.

II. Pragmatic Action Charts

PACs heavily rely on their host language. In practical Statechart projects, developers may tend to discard mealy-like transition actions all together and program their host code in the *entry actions* of *transient states* [9].

A. Model of Computation

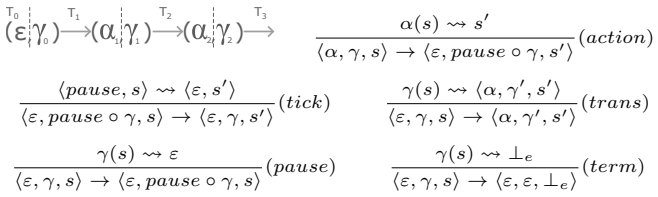
A PAC is a set of *locations* that are divided into *action* α and *control* γ. At clock tick, the PAC MoC executes the γ of the current location to transition to another location and then immediately executes the associated α of the new location. Fig. 1 exemplifies this behavior. At reset, the current location is set to (ε, γ_0) with γ_0 pointing the next location. During the first tick, the SM transitions to (α_1, γ_1) and immediately executes α_1. In the second tick, γ_1 then transitions to (α_2, γ_2) and executes α_2 and so on.

The MoC basically only consists of five rules. An action is resolved under the Host Language (HL) ($\rightsquigarrow$) producing a new program state s and issues a *pause*. A pause tick gives the program time to react to its environment. Finally, there are three possibilities for the γ control to decide in HL how to proceed: Transition to a new location, pause at the current location, or terminate the program.

All reactions take place immediately after a new location is reached. Since immediate control can directly be written in the host code components and to keep the MoC concise, there is no explicit concept for immediate action chaining on the PACs abstraction level. Same is true for cascading immediate conditions. In comparison, SCCharts [10] have the possibility to jump into these chains from any point in the automaton and PACs not. This and the lack of bi-directional communication is a trade-off towards the focus on HL integration and simplicity, which is supported by both SCCharts case studies [9], [11] w. r. t. HL integration. Since immediate reactions are intended to be programmed in host code, it seems as concept break to add immediate transitions. Nonetheless, they can be implemented as extended feature by guarding and copying the action parts of the target location.

An advantage of this separation is that it is relatively easy to implement directly in the HL using the OOP paradigm. The examples in this paper are w.l.o.g. programmed in TypeScript, because it is relatively easy to build the subsequent transient

2832-6520/24/$31.00 ©2024 IEEE
DOI 10.1109/MEMOCODE63347.2024.00020



$$\frac{\alpha(s) \rightsquigarrow s'}{\langle \alpha, \gamma, s\rangle \rightarrow \langle \varepsilon, pause \circ \gamma, s'\rangle}(action)$$

$$\frac{\langle pause, s\rangle \rightsquigarrow \langle \varepsilon, s'\rangle}{\langle \varepsilon, pause \circ \gamma, s\rangle \rightarrow \langle \varepsilon, \gamma, s'\rangle}(tick) \qquad \frac{\gamma(s) \rightsquigarrow \langle \alpha, \gamma', s'\rangle}{\langle \varepsilon, \gamma, s\rangle \rightarrow \langle \alpha, \gamma', s'\rangle}(trans)$$

$$\frac{\gamma(s) \rightsquigarrow \varepsilon}{\langle \varepsilon, \gamma, s\rangle \rightarrow \langle \varepsilon, pause \circ \gamma, s\rangle}(pause) \qquad \frac{\gamma(s) \rightsquigarrow \bot_e}{\langle \varepsilon, \gamma, s\rangle \rightarrow \langle \varepsilon, \varepsilon, \bot_e\rangle}(term)$$

Fig. 1: PAC MoC

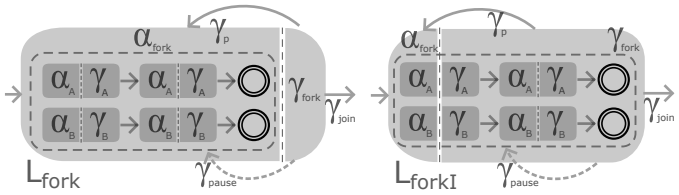


Fig. 2: _fork() vs. _forkI() semantics

views for modern IDEs using available visualization frameworks available for the web. Conceptually, the approach is not restricted to TypeScript.

B. Extended Features

Extension	Synonym	Description
_action(α)	$() \rightarrow (\alpha$, _delegate(L_1))	execute the action and reset
_control(γ)	$() \rightarrow (\varepsilon, \gamma)$	branch to location without action
_transition(L)	$() \rightarrow L$	transition curry
_root()	$() \rightarrow L_1$	jump to initial curry
_self()	$() \rightarrow$ _location(α, γ)	self loop curry
_pause()	$() \rightarrow$_control(γ)	pause curry (ε in short)
_term(e)	$() \rightarrow \bot_e$	terminate PAC curry
_if(c, γ)	$() \rightarrow c \ ? \ \gamma : \_pause()$	exec γ if c holds

Tab. 1: PACs location extensions

Several convenience methods ease the program modeling. The first two functions provide factories for action and control only locations. Furthermore, _transition points to the given location, _root leads back to the root location, and _self loops back to the calling location. _pause creates an empty control function, which does not initiates the passing of control. In contrast to _self this does not re-evaluate the α of the location and invokes γ again in the next tick. Finally, _term terminates the PAC, de facto signaling that a final state has been reached. While one can argue if a SM is considered terminated if no further behavior can be observed, explicitly marking the PAC as terminated has practical reasons especially when dealing with hierarchy. Therefore, the terminated $\bot$ state is treated as a special case with optional exit code e. The _if instruction only executes the control γ if the condition c holds. Otherwise, the location pauses.

C. Concurrency Extensions

Extension	Description
_fork(γ_p, γ_{join}, $\varphi_0 ... \varphi_{n-1}$) _forkI($\gamma_p$, γ_{join}, $\varphi_0 ... \varphi_{n-1}$)	Runs φ_n concurrently and joins to γ_{join} as soon as all φ have been terminated. γ_p may preempt the location.

Tab. 2: PACs concurrency extensions

Although PACs main purpose are not complex concurrent use-cases, the last extension, shown in Tab. 2, helps in the construction of *superlocations*. Even without concurrent back-and-forth communication within one tick, superlocations are still useful to separate logical control-flows and may pass

```
1  class ABRO extends PragmaticActionChart {
2
3    constructor(
4        readonly _inputA: () => boolean,
5        readonly _inputB: () => boolean,
6        readonly _inputR: () => boolean,
7        readonly _outputO: () => void) {
8      super();
9    }
10
11   public awaitAB(): Location {
12     return this._forkI(
13       () => this._if(this._inputR, () => this.awaitAB()),
14       () => this.doneAB(),
15       new Await(this._inputA),
16       new Await(this._inputB),
17     );
18 }
19
20   public doneAB(): Location {
21     return this._location(
22       () => { return this._outputO(); },
23       this._if(this._inputR, () => this.awaitAB())
24     );
25   }
26 }
```

Fig. 3: ABRO PAC example

information in one direction. The _fork takes two control functions, γ_p and γ_{join} and an arbitrary number of PACs, labeled φ_i. As action, the location ticks all φ_i and transitions via γ_{join} to the next location if all φ_i have terminated. γ_p is the control function of the superlocation. It can be used to preempt the action.

_fork represents a straight-forward extension if following the strict action-control separation discussed at the beginning of the section. This does, however, not suffice if one wants to implement an Esterel-like immediate *termination* if all threads have terminated. The differences are illustrated in Fig. 2. Fig. 3 shows the PAC for the synchronous *Hello World* program ABRO. Note that all convenience functions simply build upon the basic concept discussed in Sec. II-A and are written in the HL. A developer is free to add own functions and can overwrite the presented functionality.

III. Pragmatic Action Trees

This section illustrates how to extend the established MoC and extensions from Sec. II to implement Pragmatic Action Trees (PATs), an incarnation of behavior trees [12]. A Behavior Tree (BT) is a SM that is represented as a tree with two operators, *select* and *sequence*, that orchestrate actions, which are the leafs of the tree. All nodes return their current state, namely *running*, *success*, or *failure*. A selection node executes its children until a child has succeeded. It returns a failure if

none succeeds. Asymmetrically, a sequence node executes its children as long as they succeed. It returns a failure as soon as one child fails.

Extension	Description
_selector(γ_p, $\varphi_0 \dots \varphi_{n-1}$)	Runs children φ_n until $e(\varphi_n)$ succeeds. Then, terminate self successfully, or with an failure if none succeeded. γ_p may preempt the location.
_sequence(γ_p, $\varphi_0 \dots \varphi_{n-1}$)	Runs children φ_n until $e(\varphi_n)$ succeeds. Then, terminate self successfully, or with an failure if none succeeded. γ_p may preempt the location.
_selectorCtrl (γ_p, γ_{succ}, γ_{fail}, $\varphi_0 \dots \varphi_{n-1}$)	Similar to _selector but give the control to γ_{succ} or γ_{fail} respectively
_sequenceCtrl (γ_p, γ_{succ}, γ_{fail}, $\varphi_0 \dots \varphi_{n-1}$)	Similar to _sequence but give the control to γ_{succ} or γ_{fail} respectively
_immediate(L)	Immediately executes α_L and returns γ_L.

Tab. 3: PAT extensions

Since PATs are entirely built upon PACs using the same MoC, one can mix both notions. Fig. 4a shows the combination of ABRO and the DrinkTask examples. As depicted in Fig. 4b, the PAT behaves like ABRO at the start, waiting for A and B concurrently. Then, after both signals turn to true, the machines traverses to the doneAB location, where the root of the BT is located. From here, the BT executes similar to the previous DrinkTask example in Fig. 4. The whole behavior is reset as soon as R becomes true.

To achieve this behavior, two (resp. three including the selector) further extensions are added. First, _selector and _sequence terminate with the corresponding exit code as soon as their children terminate. Therefore, two new versions of _selector and _sequence redirect their control according to the exit code instead of simply terminating. Second, since the control of all locations only reacts in the following tick, the basic PAC MoC finally has reached its limits, because the BT could only react in the next tick under these circumstances. The argument from Sec. II that immediate reaction should reside either in α or γ does not hold since the locations here serve pure structuring reasons. This would be also true if one chains any *super locations* that should react immediately. Hence, we finally introduce the _immediate extension making it available for the developer, seen in Tab. 3 and used in Line 13 in Fig. 4a. _immediate immediately executes the α and returns the γ from its target location. As usual w. r. t. the Synchronous Hypothesis, it demands an acyclic path so that the macro tick can eventually finish its computation.

IV. Tool Integration

PACs are literally *executable documentations* [13]. A visual representation of the SM can be displayed instantaneously using transient views [14] with (optional) automatic layout. Fig. 5 shows an instance of the VS Code IDE. When compared to the code, one recognizes that the path from the select location to black is missing. Hence, the location is unreachable and marked as such in red. While such oversights might be difficult to spot in pure host code, the visualization makes this rather obvious.

```
1  class ABRDrinkTask extends PragmaticActionTree {
2
3    constructor(
4      readonly _inputA: () => boolean,
5      readonly _inputB: () => boolean,
6      readonly _inputR: () => boolean) {
7      super();
8    }
9
10   public awaitAB(): Location {
11     return this._forkI(
12       () => this._if(this._inputR, ()=>this.awaitAB()),
13       () => this._immediate(() => this.doneAB()),
14       new Await(this._inputA),
15       new Await(this._inputB),
16     );
17   }
18
19   public doneAB(): Location {
20     return this._sequenceControl(
21       () => this._if(this._inputR, ()=>this.awaitAB()),
22       () => this._pause(),
23       () => this._term(),
24       this.Fridge(),
25       this.Drink());
26   }
27
28   protected Fridge(): PragmaticActionTree {
29     return new class extends PragmaticActionTree {
30       public fridgeDoorOpen(): LocationFn {
31         return this._selector(
32           () => this._pause(),
33           new Failure("Fridge open?"),
34           new Success("Open fridge"));
35       }
36     }
37   }
38
39   protected Drink(): PragmaticActionTree {
40     return new class extends PragmaticActionTree {
41       public doDrink(): LocationFn {
42         return this._sequence(
43           () => this._pause(),
44           new Success("Drink"),
45           new Success("Close fridge"));
46       }
47     }
48   }
49 }
```

(a) ABRDrinkTask PAT example in TypeScript

```
A: false, B: false, R: false
A: true, B: false, R: false
A: true, B: true, R: false
Fridge open?
Open fridge
Drink
Close fridge
```

(b) ABRDrinkTask PAT output

Fig. 4: ABRDrinkTask PAT example

Finally, if one wants to create an own external DSL for their language, for example to add more convenience or to allow real polyglot models, I recommend the use of the off-side rule. The off-side rule is compatible with HLs without the use for exotic delimiter, escape characters, and languages that use the rule themselves. Off-side rule languages sometimes tend to get untidy when getting large, but since the host code in Statecharts-like languages tends to be rather small, this seems to be less of a problem. Fig. 6 shows the Colors PAC of Fig. 5 with the black color variant in an own external DSL. The PAC

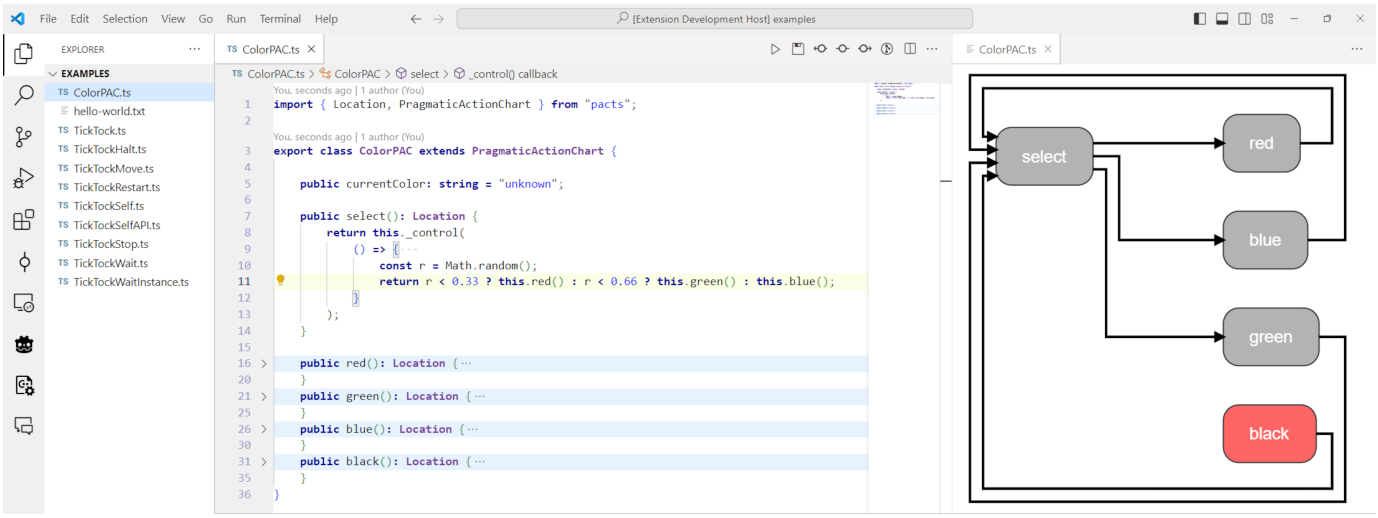


Fig. 5: Representing the current PAC as graphical model in a dedicated view in VS Code

```
pac Colors
    public currentColor: string = "unknown";

loc select control
    const r = Math.random();
    return r < 0.25 ? this.red()
        : r < 0.50 ? this.green()
        : r < 0.75 ? this.blue()
        : this.black();

loc red action
    this.currentColor="red";

loc green action
    this.currentColor="green";

loc blue action
    this.currentColor="blue";

loc black action
    this.currentColor="black";
```

Fig. 6: The Color PAC example in an external PAC DSL

begins with the pac keyword and its id. The locations are defined by loc and similar convenience methods as before, such as action and control, are introduced. As suggested, the HL of each location is indented to avoid any confusion between DSL and HL. The code depicted in Fig. 6 can relatively easy be translated into the TypeScript code shown in Fig. 5.

V. Related Work

Closely related to this integration approach is the SyncCharts [3] implementation by von Hanxleden [15]. In his work, the C pre-processor is leveraged to provide a full SyncCharts implementation within the HL. While this approach comes close to an internal Statechart DSL, the Esterel-like structure obfuscates the otherwise clear automaton structure of Statecharts. Other approaches, such as Eckel [16] in Python or cmars statechart for Rust, model the entire SM following the OOP (resp. trait) paradigm. Here, all SM elements, especially atomic entities, such as states, are modeled as classes. While precise and expressive, this may be to complex for small size projects. Statecharts languages differ especially when it comes to the host code handling. Schulz-Rosengarten et al. [17] use the OOP paradigm to organize host code in methods. Additionally, SCCharts [10] are modeled textually but are also displayed graphically instantaneously, which was adopted by PACs in a generic way allowing an appropriate filtering of internal DSLs within the HL. Similarly, Lingua Franca [18], focusing on a more dataflow-oriented and distributed approach, uses the same transient view technology to depict the interconnections between their atomic reactors. The atomic reactor entity itself is programmed in the HL, which is enriched by command of the runtime environment to enable the orchestration.

Analogously to the discussed Statechart dialects, PACs are clocked and follow the synchronous hypothesizes [19]. Usually, behavior trees [12] employ a run-to-pause semantics, which is related to the run-to-completion semantics of the UML [20]. The goal of simplicity is also driven by the general notion of language-driven engineering [21], [22].

VI. Conclusion

Pragmatic Action Charts are a minimalist Statecharts variant that is implemented as internal DSL. However, modern mainstream languages arguably lack extension methods to achieve the same form of convenience as their external DSL cousins. One idea is to provide a generic mechanism to add keywords, such as async and await in JavaScript, which basically provide convenience for the developer when dealing with nested callbacks. Basically, all extension methods presented throughout the paper are candidates for convenient keywords. As demonstrated, behavior trees can be implemented using the same approach. The exemplary TypeScript implementation of PACs and PATs is available as open-source project[1]. It serves as inspiration and a blueprint for other language extensions.

[1]https://github.com/pragmatic-programming/pacts

REFERENCES

[1] B. Selic, "A systematic approach to domain-specific language design using uml," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*. IEEE, 2007, pp. 2–9.

[2] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *Seminar on Concurrency, Carnegie-Mellon University*, ser. LNCS, vol. 197. Springer-Verlag, 1984, pp. 389–448.

[3] C. André, "Representation and analysis of reactive behaviors: A synchronous approach," in *Computational Engineering in Systems Applications (CESA)*. Lille, France: IEEE-SMC, July 1996, pp. 19–29.

[4] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[5] M. Broy, K. Havelund, R. Kumar, and B. Steffen, "Towards a unified view of modeling and programming (track summary)," ser. Lecture Notes in Computer Science, vol. 9953, 2016, pp. 3–10.

[6] S. Bonfanti, M. Carissoni, A. Gargantini, and A. Mashkoor, "Asm2c++: a tool for code generation from abstract state machines to arduino," in *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9*. Springer, 2017, pp. 295–301.

[7] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[8] V. Kushnir and B. Koman, "Creating ai for games with unreal engine 4," *Electronic Journal of Information Technology*, vol. 9, 2018.

[9] S. Smyth, C. Motika, A. Schulz-Rosengarten, S. Domrös, L. Grimm, A. Stange, and R. von Hanxleden, "SCCharts: The mindstorms report," Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1904, December 2019, ISSN 2192-6247.

[10] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[11] S. Smyth, C. Motika, A. Schulz-Rosengarten, N. B. Wechselberg, C. Sprung, and R. von Hanxleden, "SCCharts: the railway project report," Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Technical Report 1510, August 2015, ISSN 2192-6247.

[12] K. Winter, "Formalising behaviour trees with csp," in *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings 4*. Springer, 2004, pp. 148–167.

[13] S. Smyth, J. Petzold, J. Schürmann, F. Karbus, T. Margaria, R. von Hanxleden, and B. Steffen, "Executable Documentation: Test-First in Action," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2022, pp. 135–156.

[14] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model! – Putting automatic synthesis of node-link-diagrams into practice," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA: IEEE, September 2013, pp. 75–82.

[15] R. von Hanxleden, "SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency," in *Proc. Int'l Conference on Embedded Software (EMSOFT '09)*. Grenoble, France: ACM, October 2009, pp. 225–234.

[16] B. Eckel, *Python 3 Patterns, Recipes and Idioms*. readthedocs.org, 2008.

[17] A. Schulz-Rosengarten, S. Smyth, and M. Mendler, "Towards object-oriented modeling in SCCharts," in *Proc. Forum on Specification and Design Languages (FDL '19)*, Southampton, UK, September 2019.

[18] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 4, pp. 1–27, 2021.

[19] D. Potop-Butucaru, R. De Simone, and J.-P. Talpin, "The synchronous hypothesis and synchronous languages," *The embedded systems handbook*, pp. 1–21, 2005.

[20] B. Selic, "Uml 2: A model-driven development tool," *IBM Systems Journal*, vol. 45, no. 3, pp. 607–620, 2006.

[21] T. Margaria and B. Steffen, "Simplicity as a Driver for Agile Innovation." Los Alamitos, CA, USA: IEEE Computer Society, 2010, vol. 43, no. 6, pp. 90–92.

[22] B. Steffen, F. Gossen, S. Naujokat, and T. Margaria, "Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages," in *Computing and Software Science: State of the Art and Perspectives*, B. Steffen and G. Woeginger, Eds. Springer, 2019, vol. 10000.