ing these properties. In that, CP shares many parallels with reactive programming. Consequently, we consider interactive CP systems being reactive in this thesis. Interestingly, integrations of CP systems into OO host languages became more and more sophisticated over the years, culminating in Babelsberg. Babelsberg makes CP more accessible to OO developers by reducing the knowledge on CP required. To do so, Babelsberg allows developers to make full use of familiar OO concepts to describe constraints and reduces the configuration overhead by following the convention over configuration principle.

⁂

A large landscape of varied concepts, working principles, and systems emerged from the basic idea of reactivity. In particular, systems differ in their applied data structures, dependency creation process, change triggers, type of reactive behavior, and consistency guarantees. Note, that the presented concepts and systems are only a selection of the available design space. The general notion of reactive programming spans multiple distinct areas of research and research communities from different backgrounds, each with their own perspectives, values, and application domains. Consequently, even when we encounter recurring concepts, such as events, behaviors, or constraints, they are interpreted in a slightly but distinctly different manner. Thus, even when different reactive programming concepts can solve similar problems, they are not arbitrarily exchangeable linguistic tools. Instead, the concepts we use influence our perception of and reasoning about the problem at hand.

2.3. Change Detection and Reaction as Coarse-grained, Unified Model for Reactive Programming

To support the development of a wide range of reactive programming concepts, active expressions require us to identify some commonality among reactive programming concepts to leverage for reuse. Unfortunately, reusable mechanisms are hard to come by given the diverse nature of reactive behavior introduced in Section 2.2. Even worse, many concrete implementations of reactive programming concepts are rather monolithic in order to maximize ease of use and performance. As a result, reactive programming concepts are not designed for reuse with missing extension points making it hard to chop a reactive programming concept into pluggable blocks.

Despite these rigid characteristics, some researchers acknowledge the diversity of reactive programming concepts and made attempts towards a common taxonomy that spans multiple reactive programming families. For example, Margara and Salvaneschi provide a five step model (observation, notification, processing, propagation, and reaction) for signal systems and CEP [154]. However, because this taxonomy is designed to support signal systems and CEP in particular, it is too narrow to be directly applicable to all reactive programming concepts. To be precise, simple reactive programming concepts skip some steps entirely, e.g. a one-way data binding skips the processing step, while complex ones

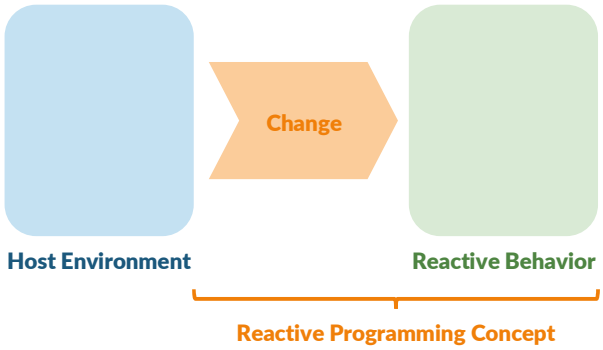

Figure 2.11.: Reactive programming propagates change from one part of the system to another.

do not strictly follow the model, e.g. Wallingford would require two such processes, one for time advancement and one value queries, each with different properties. Furthermore, as the taxonomy is intended for conceptual comparison, it does not present its steps as pluggable extension points for behavior variations.[28]

Given the vast variety of reactive behavior, it is hard to find common model that covers most reactive programming concepts well. The more fine-grained a model the more sparse is its feature matrix and the higher the chance to encounter a system that challenges the model's fundamental assumptions in unique and unexpected ways as illustrated above.

Rather than introducing such a fine-grained model, we aim for a simpler and overall more inclusive approach: instead of dissecting highly different behaviors into buckets of common steps, we acknowledge their incompatibility. To still attain a common perspective on reactive programming, we abstract this variety out of the formula. That means, we handle the type of reactive behavior executed as a single (big) variation point. To continue our reasoning, if these varying behaviors are not essential to reactivity itself, what remains is at the very heart of reactive programming: the very concept of *change*, its *detection* and *propagation*. To elaborate, we argue that the notion of reactivity is not about any kind of reactive behavior itself, but about this *behavior being executed*. Following our definition of reactive programming from Section 2.1, which states that reactive programming establishes an ongoing connection between multiple parts of a program, we see reactive programming as a *glue paradigm*. Rather than enforcing a particular execution model, reactive programming connects execution in one paradigm (the host) to another (the reaction) via the propagation of change as depicted in Figure 2.11 [234]. During execution of the host, some interesting change happens, a reactive programming runtime detects this change and propagates it to interested parties, ultimately, entailing some form of reaction. Given these considerations, reactive programming concepts adopt a *two part structure* of with the first part handling change detection and the second handling the reactive behavior itself. For this thesis, we focus on OO imperative programming as the primary paradigm of the host language. The second part's paradigm may or may not coincide with the first one.

[28] Section 7.1 compares this taxonomy with ours in more depth.

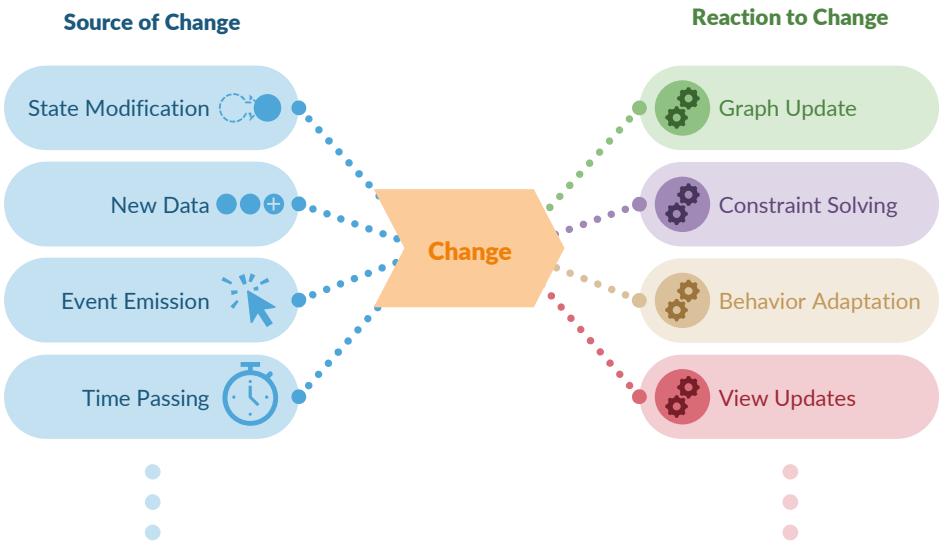

Figure 2.12.: Different types of sources (left) of and reactions (right) to change are tied together by their shared understading of the concept of change.

This perspective on reactive programming also explains the perceived variety of reactive programming as reactions may come from any programming paradigm. For example,

- interactive constraint systems solve given constraints,
- implicit invocation systems call multiple imperative procedures, and
- signal systems update the values in their DAG.

While being diverse, one quality most reactions have in common is that they execute behavior *with respect to the change that triggered them*. To expand our examples, consider that

- constraints are not simply solved but incorporate a temporary assignment constraint to account for the change,
- procedures are called with arguments specific to the change, and
- signal systems only need to update the part of their DAG that is affected by the change.

Not only the reaction to change presents a variation point for reactive programming concepts but source of a change may vary as well. Examples for such sources include state modifications, event emissions, new data, or elapsed time. As depicted in Figure 2.12, all these sources lead to some form of change that is then propagated.

Various combinations of source types and reactions are possible. For example, many libraries for interactive constraint solving require an explicit notification on constraint violations while OCP infers these violations implicitly from state modifications in the underlying runtime. Nevertheless, both variants result in some form of change propagation through constraint solving. Similar, a single source of a change may trigger different update mechanisms. For example, an event emission might trigger imperative computation in an

implicit invocation system, cause instantaneous changes in a continuous time system, or the next iteration in a synchronous dataflow graph.

2.4. Selected Dimensions of Reactive Programming Concepts

Given the two-part structure for reactive programming concepts identified in the previous section, we investigate individual property dimensions of these parts to further improve our understanding of potentially reusable components. To be specific, we highlight three selected dimensions of reactive programming concepts, namely the type of change notification, how cascading reactive behavior is handled, and the embedding of reactive programming into the host language. We selected these dimensions in particular, as they provide design perimeters for a reusable component, either by narrowing down design space because we see one specific approach in the dimension as desirable or by illustrating the range of a dimension that we have to account for.

2.4.1. Explicit and Implicit Change Notification

In the previous section, we briefly highlighted various sources of change, such as state manipulation and event emission. One dimension of reactive programming concepts regards whether change is explicitly or implicitly detected. To be specific, this dimension specifies who is responsible for detecting and notifying about a change, the reactive system or the developer. We can easily distinguish these two cases by checking the source code: explicit change notifications are manifested in source code while implicit ones are not.

Name for trigger: source of a change/Impulse/Trigger/Auslöser/Activator/Initiator/Notification/Observation

Explicit Change Notification To illustrate the difference, consider the following example of an *explicit change notification* for a state manipulation as often seen in the observer pattern:

```
this.x = 5
this.notify()
```

The first line handles the actual state manipulation in the host language. The second line contains the explicit change notification, an instruction *in the source code* with the sole purpose of signaling a change in the program to the reactive programming system, thus, instructing it to respond to and propagate the change. For explicit change notification to be used, the change has to be detected previously by other means. In our example, the application developer simply knows about the change in the previous line and uses a dedicated statement for notification purpose only to inform the reactive programming system about it. Some sources of change lend themselves well to explicit notification, including event emission in a publish-subscribe system (`emit('drag', { x: 50, y: 50 })`) or pushing new values into a dataflow stream (`subscriber.next(42)`).

Implicit Change Notification Implementing the same example with an *implicit change notification* results in the following code: