A Reactive Specification Formalism for Enhancing System Development, Analysis and Adaptivity

Assaf Marron
Weizmann Institute of Science
Rehovot, Israel
assaf.marron@weizmann.ac.il

1 INTRODUCTION

During system development, external reviewers, especially ones with expertise in the problem domain or in system and software engineering (SE), often contribute insights that up to that point were not noticed by the engineers and other project stakeholders. These reviewers apparently do so by employing special human competencies that presently are not, or even cannot be, automated. Consider, for example, the competencies that enable the following review comments about a home-assistant robot: (a) during code review: “I see that the robot can defers command execution until it charges its battery for the task; When done charging, does the robot check if the action is still needed?”; (b) following a demo: “Will the robot trip over a thin transparent phone cord?”; and (c) “Some clear voice commands had to be repeated. Perhaps sometimes the robot isn’t listening?”.

One of the aims of our wise computing vision and research [1–3] is the incremental automation of such competencies. These include causal analysis, prediction, observing emergent or unspecified patterns, and, assessing whether these are desired or not. Moreover, humans with varied skills and backgrounds can discuss such findings and their handling. Automation will enable exercising these analyses constantly and repeatedly throughout the development cycle over the entire set of project artifacts: documents, code, live and recorded demos, etc. In this paper we describe one aspect of our wise-computing research, namely, the search for a common formalization of reactive behavior, hierarchy, orthogonality and concurrency. Together with encapsulating environment interfaces in sensors and actuators programmed in any language, this makes the specification executable — as part of the system or in automated analysis tools.

At the core of CF1 is the behavior-composition semantics of scenario-based programming (SBP), a.k.a. behavioral programming (BP) (see review paper [4] including the references and related work therein). This enables composing complex systems from concurrent, interwoven execution (play-out) of seemingly-stand-alone scenarios, as derived from, e.g., requirements, use-case descriptions, or user manuals. Similarly, system analysis scenarios can be gleaned from reviewers’ explanations of how they conduct a review and what they look for. E.g., SBP enables composing partial scenarios like “When driving towards an intersection, if the traffic-light is red, stop.” and “but if a police-person motions you to keep driving, do not stop. Indeed, a police instruction overrides everything else.” and “but, still, if a running pedestrian is about to cross your path, do nevertheless stop!”. With wise computing, we aim to automate the application of world knowledge (e.g., “keep people safe”) and results of system analysis (e.g., “an accident is still possible”) towards identifying, say, the need for the third scenario, after the first two were coded. And, we wish to do so using heuristics and abstractions like those that enable humans to notice and handle such risks quickly, without exhaustive exploration all possible system behaviors and states.

Comparison of CF1 to other formalisms is presented in Section 5.

2 THE CF1 FORMALISM

Principles. Each basic element of CF1 should: (a) be stand-alone: i.e., not depend on many other knowledge entities (consider, e.g., automating a technique that an expert reviewer used and (b) be composable, e.g., new facts and rules should be able to either be orthogonal to, or override and refine others, either implicitly (referencing common actions or conditions) or explicitly (referencing specification elements); (c) handle in similar ways minute details and abstract concepts (e.g., criteria for detecting a wait command that lacks timeout handling, an unintuitive user interface widget, or a hard-to-maintain overall system architecture. (d) be executable, towards automating knowledge use in SE; (e) be associative and connectible, e.g., when automating the detection of end-user actions that lack immediate feedback, encode the facts that pressing a keyboard key, swiping a touch screen, turning a lever arm, and uttering a voice command can be end-user actions, and that click sounds, display changes, lights, and tactile effects can be feedbacks.
The basic idioms. A CF1 specification is a highly-connected network of basic computational units (BCUs). Each BCU captures the essence of a single concept or concern, like a program instruction, a design principle, or a tip for creating tricky test cases.

The BCU is a ‘device’ built as a minimalist state machine. It has exactly two states, T and F, and two transitions, F-to-T and T-to-F. Given a BCU $u$, we denote its states as $u^T$ and $u^F$, and generally as $u^s$, with $s \in \{T, F\}$. Each transition has a name, $u^s$, which is the same as its target state. There are no self transitions, and no other outgoing transitions. Every BCU state is labeled with two sets, $R$ and $B$, of transition names. $R$ lists requested transitions (in other BCUs), and $B$ lists blocked transitions. The BCU composition and execution semantics follows the SBP/BP principles [4], and is explained in Fig. 1 and in the semantics subsection below.

![Figure 1: A BCU example. BCU $u_0$ is in its $T$ state ($u^T_0$, in bold), requesting the transition of $u_1$ into its $T$ state, and blocking the transitions of $u_5$ and $u_6$ into $u^T_5$ and $u^T_6$, respectively. $u_0$ itself will transition from $u^T_0$ into $u^F_0$, when $u^T_6$ is in the $R$ set of some current BCU state and is not in the $B$ set of any, and the event is selected from all enabled ones.](image)

Transition Semantics. One of the two states is designated as the BCU’s initial state. In the initial system configuration each BCU is in that state. A transition is enabled if (a) the BCU is in the transition’s source (= ‘from’) state; (b) it is in the $R$ set of the current state of some BCU; and, (c) it is not in the $B$ set of any of the current BCU states. In other words, a transition can be triggered if it is applicable, if at least one BCU proposes (i.e., requests), that it be considered for triggering, and if no BCU forbids (i.e., blocks) its triggering.

The notation and semantics of the $R$ and $B$ sets can also be viewed as SC-like concurrency guards on BCU transitions. A transition $u^s$ thus depends on the system being in at least one of a BCU state (those that request $u^s$), and in no state from another set (those that block $u^s$). We use these two views of CF1 interchangeably (various graphical and textual notations for CF1 concepts are being explored).

The direct execution of a CF1 specification progresses via state-transition steps. In each step one enabled transition is selected (randomly or following a strategy); the relevant BCU switches states (i.e., the transition event occurs); and, the next step of finding and triggering an enabled transition takes place.

As in SC and SBP/BP, we adopt the formal logical-execution time model of Henzinger et al., where no two events occur simultaneously and system-driven events take zero time. Also, as in SC and SBP/BP, BCUs can be organized in mutually-independent scheduling components or objects, where transition-selection and triggering can occur asynchronously across such components.

Sensors and Actuators. Sensors and actuators are BCU variants that are connected to the physical environment and represent real-world, environment concepts. A sensor (resp. actuator) state can be simple, like reporting that a user has clicked a button (resp. actuating the physical turning-on of a light), or complex, as in visually sensing that another car is approaching (resp. actuating the turning of a car’s wheel). By convention, a sensor’s transition is never blocked, and an actuator performs its action either when it transitions to a corresponding state, or constantly when in that state. The implementation of sensors and actuators can use arbitrary techniques. These include, e.g., using classical programming to access a device’s API to switch on a light, using machine learning to sense that an object in the camera view is a moving vehicle, or, further using CF1 specifications to combine lower level sensors and actuators into richer ones. Sensors can be devised also for complex self-reflective observations about the system itself, for SE insights (e.g., detecting a loop or the absence of required exception handling), or for application-specific ones (e.g., jerk robot motions, or unjustified rejection of valid commands). The exact scheduling of when sensor/actuator events take place relative to other BCU transitions will be defined as part of the full formal definition of CF1, and will likely be inspired by the corresponding definitions in SC and SBP/BP. This issue is secondary, though, to CF1’s handling of composite system behavior and development processes.

The Execution Log. CF1 includes a dynamically accessible execution log (or trace) of each run, which formally and ideally captures ‘absolutely everything’. After every environment or system event, a copy of the specification, i.e., all BCUs and their current states, is saved ‘forever’. In a procedural programming analogy, this would mean keeping not only all data, but also a complete copy of the running program code (as it may change), and, the program counter of each parallel process. The goal is to save all(!) relevant details that may distinguish two similar event sequences. E.g., by recording all current BCU states, such a log captures all enabled transitions that were not selected. In practice the log size will of course be dramatically reduced via standard data compression, filtering-out of repetitive or unimportant data, replacing raw data with summaries, sharing similar records across application instances and across the world, and, subjecting the log to adaptive retention decisions. Keeping interim comprehensive snapshots followed only by a change log are also possible, as long as logged states are readily indexed and retrieved. In distributed executions, logs can be merged physically and/or virtually off-line.

Specification interfaces. A CF1 specification is a reactive system in its own right, distinct from the systems built using it. It reacts to the effects of its own behavior, and to changes in environment assumptions and in engineering needs. The system, the execution infrastructure and external tools can thus, e.g., (i) create or replicate a BCU (ii) set or modify a BCU’s $R$ or $B$ set (iii) find BCUs that have desired properties (iv) search the execution log for certain composite states and event sequences.

3 A SMALL EXAMPLE

Below is a textual description of CF1 specification for a small system, followed by specification of external knowledge used in reviewing it.

The system has two mouse buttons, $B_1$ and $B_2$, and a green and a red light named $GL$ and $RL$ respectively. When a user clicks $B_1$, a sensor BCU $u_{B_1}$ reports this by going into its $u^T_{B_1}$ state. The click sensor $u_{B_1}$ will transition back into its $u^F_{B_1}$ state after all BCUs that
waited for this click-report could have been properly notified, but before any newly-transitioned BCUs listening out for the next click can observe it. BCUs $u_0, u_1, u_2$ are arranged such that $u_i$ is in $T_i$ when the number of B1-clicks is $i$ modulo 3, and an actuator BCU flashes $GL$ when $u_0^T$ holds (once every 3 clicks). Specifically:

- $u_1^E : u_{B1}^T \land u_0^T \rightarrow u_1^T \ (1 \ text{modulo} \ 3 - \ when \ \text{BCU} \ u_1 \ \text{is in state} \ \ u_1^T, \ if \ \text{the system is in} \ u_0^T \ \text{and in} \ u_{B1}^T \ \text{then transition into} \ u_1^T)$;
- $u_2^E : u_{B1}^T \land u_1^T \rightarrow u_2^T \ (2 \ \text{modulo} \ 3)$;
- $u_0^E : u_{B1}^T \land u_2^T \rightarrow u_0^T \ (0 \ \text{modulo} \ 3)$;
- $u_{GL}^E : u_0^T \rightarrow u_{GL}^T \ (\text{actuate green light})$;
- $u_1^T : u_{B1}^T \rightarrow u_1^T \ (\text{next click makes condition false})$;
- \ldots

and so on. Similarly, another set of seven BCUs, say $u'_i$ through $u'_6$ and an actuator $u_{GL}$ are arranged to cause the flashing of the red light $RL$ every 7 button clicks.

Note that the above formal notation can be automatically translated from or to natural language sentences.

The power of composition and formal analysis in CF1 specifications (derived from that of SBP/SP, and from the separation of logic from sensors and actuators) can be seen, e.g., in the ability of formal methods such as SMT solvers to predict that both $GL$ and $RL$ will flash together every 21 B1 clicks, or, further, at the product of the two cycle lengths, even when consisting of thousands of clicks.

Incremental modular (though manual) development can be seen in the following example: assume that requirement $\tau$ specifies that when button $B2$ is pressed and held, and are both to flash together, then none of the lights should flash. BCU $u_{B2}$ would sense the pressing of $B2$; another BCU, $u_*, \text{could check that both light cycles are at states}$ where they request flash actuations, and block the latter:

- $u_2^E : u_0^T \land u_{GL}^T \rightarrow u_2^T \ (B = \{u_{GL}^T, u_{RL}^T\}$).

This CF1 model is also amenable to automated answering of developer/reviewer querying. E.g., “Are there any BCUs, other than the above two cycles, that depend on buttons B1 or B2?”; “What could cause GL actuation?”; or “why did GL and RL not flash now together after B1-clicks numbered 2121 and 1221?”. In the last question, the system can quickly show that 12121 is not a multiple of 21, and whether button $B2$ was pressed at click 2121. If the log shows that $B2$ was not pressed, it will also show which BCUs requested the flashing of the lights, which ones did not, and which ones blocked it, exposing a specification error or actuator malfunction.

Further, the diligent human reviewer above, could readily (perhaps automatically) generate a monitor that uses the assertion that both lights flash together after every 21 B1 clicks. They could do so using a new counter (independent of the programmed ones) and on a new independent visual sensor that checks that $GL$ and $RL$ are physically on. The reviewer could still plug this new monitor into the specification, e.g., relying on the existing $B1 \text{click sensor}$.

Finally, assume that the reviewer indeed found problems. Asked about how they thought about it, they might say, “Whenever I am shown only short demos that start from an initial state, I ask for a run that is at least ten times longer.”. The reviewer could also say, “I also generate misleading questions on test cases (like 12121), to see if the system’s behavior can be readily analyzed and explained.” Such general and abstract test-case generation tips could be readily translated into additional CF1-based monitors that drive the generation of the long runs, and the asking of the misleading question.

4 DISCUSSION

4.1 CF1 Traits and Capabilities

The key capabilities of CF1 include:

Intuitiveness. Given the definitions of the sensors and actuators, the meaning of every BCU should be self explanatory; the BCU name should serve only as mnemonic. Where this is not obvious, instead of writing comments, one may break down complex conditions, and separate condition evaluation from the action that they drive, e.g. one can break a BCU that encodes “if ((A and B) or (C and D)) then do E” into four BCUs, encoding separately each of the three conditions, and the fact that the third drives the action. The BCUs can also encode in consistent and integrated manner diverse concepts including requirements, actual code, domain-specific knowledge, engineering skills, problem descriptions, etc. while relying on external sensors and actuators, and on the self-reflection manifested in one BCU referencing another.

Executable and analyzability. The definitions and execution semantics clearly show that CF1 specifications can be executed by a computer and/or formally verified.

Incremental manual enhancement. As shown in Section ??., CF1 facilitates incremental automation of applying human wisdom to SE. Scenarios can even begin to synthesize specific solutions such as speaker authentication.

Reflection, causal analysis and prediction analysis. CF1’s structure and its comprehensive logging enable a vast array of automated SE reasoning competencies, like monitoring for infinite loops and deadlock conditions, suggesting user actions that can reactivated a ‘stuck’ system, or discovering why a system did not act as expected.

Run-time Adaptivity. Run-time adaptivity can be achieved using the log and controlled event selection. E.g., one may add application-specific sensors for ‘system well-being’ and for imminent risks and ‘danger’. Then, when ‘danger’ is sensed, in the absence of explicit instructions, the system will search the log for actions taken in similar states, that improved or worsened ‘system well-being’, and repeat or avoid such actions accordingly. Clearly, one can also apply machine-learning to the log to synthesize CF1 specifications for improved handling of underspecified situations.

Run-time Handling of Emergent Object Classes. CF1 allows mimicking the sophisticated adaptive process of determining the existence of new, never-before seen, classes of objects, and then drawing conclusions about them, e.g., as a human would do when first encountering a new invention without any explanation. CF1 facilitates this via the direct access to sensor information (physical and computed) without object reference. Automated processes can then find behavior and relationship patterns across such properties, infer new class concepts, and use their similarities to others in automated reviews and in adaptation.
4.2 Formalism development considerations

Questions and factors in experimenting with CF1 include:

Minimalism vs. Convenience. Should the formalism have all the idioms needed by engineers, or should it be minimalist and only serve as target ‘machine-code’ into which one compiles knowledge specified in, say, Java, UML, or Statecharts? E.g., connected BCUs can readily represent data bits, text and numeric fields, complex objects, and arithmetic computations. What data structures and manipulations should be implemented as part of CF1? The same question holds also for managing context and BCU-to-BCU relationships like abstraction, set membership and containment.

Alternative BCU definitions. Should BCUs have more states? Or less (e.g., without R and B sets in an F state)?

Representation. What should the textual and visual notation be, and how should one view and navigate CF1’s rich connectivity?

Functions and parameters. Several solutions exist in CF1 for executing a particular computation on multiple objects. E.g., sensors and actuators can seek out their changing targets (like file records), or specification segments can be replicated and dedicated to specific objects. Should the concept of a function be added?

Scalability. We are confident that scalability issues in large CF1 specifications can be manageable because: (i) systems with millions of lines of code prove practical today; (ii) A wise-computing development environment can be valuable even of its expertise is limited; (iii) Like SBP/BP, CF1 execution can be distributed and parallelized; (iv) Some CF1 concepts appear to be aligned with certain brain functions. E.g., a single neuron can represent a complex concept (see, e.g., Fried et al.), and memory recall can produce experiences nearly identical to the original ones. As neurons, which are slower than an average computer circuit, can form large networks that perform complex tasks quickly, this suggests that computerized execution of CF1 specifications can be scalable as well.

Time. The concept of time can be incorporated into CF1 specifications via time-tick sensors. Should it be inherent in the formalism?

Encapsulation. CF1’s high connectivity may appear to conflict with the SE pursuit of modular encapsulation. However, we believe that with a proper methodology, such added connectivity will prove to represent important undocumented assumptions, like when a method does not check its parameters as it relies on its caller to do so. In CF1 we expect this assumption to be encoded.

Maintainability. With everything connected, and every state named and exposed, new additions to a CF1 specification might trigger the question of how they relates to each of the existing ones. We believe that first, contextual organization would help mitigate this issue, and, second, that the question deserves its own research agenda in many existing development situations considering, e.g., specification impact analysis, methodological checklists and a multitude of safety and security regulations.

5 RELATED WORK

Clearly, CF1 principles can be casted in other languages and formalisms including state machines, statecharts, scenario-based/behavioral programming, temporal logic, BIP, the B method and Z notation, CCS, CSP, synchronous languages, aspects, actors, agents and others. A key capability of CF1 is the composition of stand-alone, partial pieces of knowledge that use varied terminologies and abstractions, into an integrated specification, in a way is executable, verifiable, and intuitive to development professionals. More specifically, CF1 extends certain existing formalisms as follows:

Differences from native FSM and SC include CF1’s the compositionality of SBP/BP enabled by requesting and blocking events, and the removal of all other ways of triggering events in SC. BCU connectivity can extend Statecharts’ single state-containment hierarchy, allowing multiple containments of a single entity which enables highlighting fine commonalities and differences between entities, unbounded refinement and abstraction, and subjecting a behavior to multiple orthogonal contexts. CF1 complements this SC enhancement with extending SBP/BP in forcing scenario states to be visible rather than hidden, making it possible to condition any BCU transition on any BCU state. This is possible in SC at run time, but in CF1 also in development, for causal analysis or for meta operations such as enhancing all scenarios having states with a certain property.

A highly connected CF1 model is different from that of common neural nets and deep learning in that its depth or degree does not have to be constrained and in that external inputs and outputs can be fed at any node. Further, each BCU in a CF1 specification carries a well-defined meaning, as implied by its direct an indirect connections. In parallel, the latest deep learning techniques can be used by CF1-based systems and wise computing IDEs in SE sensors and actuators.

Compared with procedural languages like C++, CF1 specifications are more amenable to distributed execution, and can more readily access the execution log. CF1 complements knowledge representations based on logics or on ontologies by being directly executable as part of a final application or of automated analysis tools. Finally, CF1 differs from hardware chip design in that any signal (BCU state), can be readily made to depend on the state of any other signal, and selecting an event from all enabled ones can be subjected to an elaborate strategy. For more comparisons see also [1–4]).

6 CONCLUSION AND FUTURE WORK.

We presented an emerging formalism to help automate human-like competencies in development environment. We are pursuing demonstrating its application on a SC and SBP foundation. Work also continues on complementary pillars of wise-computing including an analytic engine and intuitive human interfaces. CF1’s apparent alignment with certain brain functions may help in brain modeling and may inspire enhancements to the formalism itself.

Acknowledgment. This research was funded in part by grants from the Israel Science Foundation, Minerva Foundation and German Israeli Foundation (GIF). Many thanks to David Harel, Guy Katz, Gera Weiss, Ariel Rosenfeld, Achiya Elyasaf, Daniel Yellin, and the anonymous reviewers, for valuable comments and suggestions.

REFERENCES