Labor Division with Movable Walls: Composing Executable Specifications with Machine Learning and Search (Blue Sky Idea)

David Harel,[1] Assaf Marron,[1] Ariel Rosenfeld,[1] Moshe Vardi,[2] Gera Weiss[3]
[1]Weizmann Institute of Science, Rehovot, Israel.
[2]Rice University, Houston, Texas, USA.
[3]Ben-Gurion University, Beer-Sheva, Israel.

Abstract

Artificial intelligence (AI) techniques, including, e.g., machine learning, multi-agent collaboration, planning, and heuristic search, are emerging as ever-stronger tools for solving hard problems in real-world applications. Executable specification techniques (ES), including, e.g., Statecharts and scenario-based programming, is a promising development approach, offering intuitiveness, ease of enhancement, compositionality, and amenability to formal analysis. We propose an approach for integrating AI and ES techniques in developing complex intelligent systems, which can greatly simplify agile/spiral development and maintenance processes. The approach calls for automated detection of whether certain goals and sub-goals are met; a clear division between sub-goals solved with AI and those solved with ES; compositional and incremental addition of AI-based or ES-based components, each focusing on a particular gap between a current capability and a well-stated goal; and, iterative refinement of sub-goals solved with AI into smaller sub-sub-goals where some are solved with ES, and some with AI. We describe the principles of the approach and its advantages, as well as key challenges and suggestions for how to tackle them.

Introduction

Artificial intelligence (AI) techniques are increasingly proving their power and usefulness in implementing real-world systems — resulting in a growing use of components based on machine learning, planning and search, multi-agent systems, a host of heuristics-based techniques, etc. In another area of software and system engineering, certain specifications and modeling formalisms, like Statecharts (Harel 1987) and scenario-based programming (Damm and Harel 2001; Harel and Marelly 2003; Harel, Marron, and Weiss 2012), which describe separately each facet of the desired system behavior, are also directly executable. Such formalisms, collectively referred to here as executable specifications (ES) can endow systems with often-absent properties. These include, e.g., intuitiveness of the specifications, alignment of the system's structure with the requirements, and compositionality/incrementality, i.e., the ability to enhance or refine a system by simply adding modules, similarly to how one can enhance a requirements document by adding clarifications, refinements and exceptions in the form of new sentences in the body of the document or as independent appendices and footnotes. Another important property is the amenability of ES to formal verification and analysis. This includes, e.g., model-checking in search for undesired behaviors (Harel et al. 2013; Baier and Katoen 2008) and realizability checking—looking for inherent conflicts within the specifications (Greenyer et al. 2016; Vardi 1995). In this paper we outline a novel approach for integrating the use of AI and ES techniques in developing complex intelligent systems, which can result in systems that are much easier to enhance and maintain compared with current uses of AI in system development.

Basic Principles of the Approach

Below we list the basic general principles of the proposed approach. Further illustration, explanations and examples appear in Fig. 1 and in subsequent sections.

First, one must state the overall problem or mission as a set of explicit goals, where the recognition of whether they are achieved or not in a given system run can be automated. While automatic detection of mission-accomplishment at run-time is valuable, the approach initially suffices with requiring it only at development time.

Second, when such a test or verification fails, one must be able to succinctly describe the difference, or gap, between what the system does and what it should do, generalizing a single or few bad runs into the common characteristics of all bad runs that could be associated with the noticed failure.

Third, as goals are refined and requirements added, or when certain bugs are discovered, rather than enhancing and often complicating existing components to address the issue, we strive to add new components that precisely address the difference, or the gap, between the (revised) goals and the what the existing specification accomplishes.

Fourth, we aim at an architecture in which independent ES-only components can be readily composed with independent AI-only components, where each AI or ES component addresses a particular (sub-)goal, and each is essentially unaware of the existence of the others.

The fifth principle calls for repeatedly identifying opportunities for replacing, at least partially, AI-based components with ES ones. Such transitions from AI to ES can be triggered by, e.g., the emergence of a better understanding

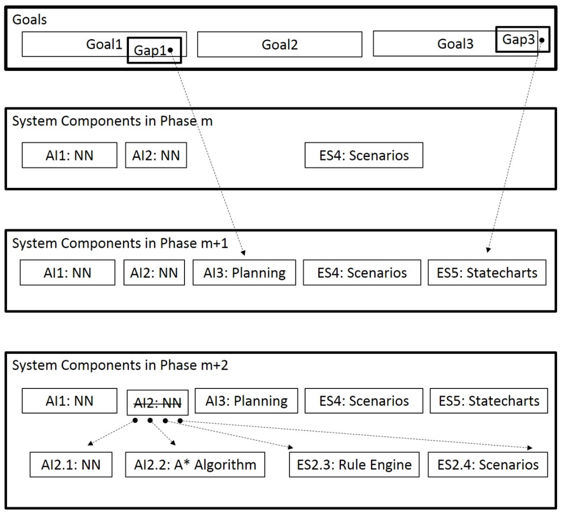

Figure 1: Progression of system structure: As Gap1 & Gap3 are discovered in verification against goals, new components AI3 and ES5 are developed to respectively address these gaps. Then, the neural net in AI2 is replaced by executable rules and scenarios, and by another neural net and a search for two remaining subproblems.

of the elements of a particular goal by engineers and stakeholders, a demand by engineers or regulators for better visibility of the logic or rationale of a specific system behavior, or a performance issue that must be corrected. For illustrating the principles, in Fig. 1, AI component AI2, based on a neural net, is replaced by two ES components ES2.3, ES2.4 containing rules and scenarios, and sub-problems previously covered by AI2 that are not addressed by ES2.3 and ES2.4, are solved by a new neural net and an A* search in components AI2.1 and AI2.2.

Goal Specification and Recognition

Complex intelligent systems, and reactive systems in general (see (Harel and Pnueli 1985)), are normally developed incrementally, using established software-engineering techniques, in a process where goals are set, and behavior is specified, implemented, and verified against those goals. The *goals* of a system (or component thereof) describe its ultimate functional requirements, or, in other words, 'what the users, customers, engineers or other stakeholders really want'. Such requirements are usually documented in texts and diagrams, and then implemented in code. The code for a given requirement is sometimes considered the ultimate formalization of that requirement. Nevertheless, because the code is often arcane and might be spread across multiple modules, there is a need to formalize the requirements themselves. Although this is sometimes done in a purely declarative fashion, *executable specifications* constitute a formal system description, whose parts describe the various aspects of the system's behavior, and whose semantics enables direct execution of the desired behavior, most often without involving computationally-expensive synthesis.

Some kinds of executable specifications, e.g., the *play-out* technique for the LSC language (Harel and Marelly 2003), or the event-selection mechanism of behavioral programming in Java or in C++ (Harel, Marron, and Weiss 2012), are characterized by the ability of the execution engine to continuously consult the constraints implied by the specifications, in an attempt to determine a valid next action for the system to take that does not violate those constraints. Even so, the development artifacts of complex reactive systems normally prescribe local reactions to events and conditions, and not the pursuit of overall system goals as is often the case in AI-based systems. Look-ahead search techniques in ES, as in *smart play-out* (Harel et al. 2002), can sometimes help choose among multiple options allowed by the specification to within a limited depth/horizon (i.e., looking ahead some fixed number of steps, while taking into account all possible responses by the environment, even an adversarial one). Complete, computationally expensive (and often impractical) program synthesis is, however, needed in order to generate an event selection strategy that copes with any environment behavior at any depth.

Thus, if the system is developed completely with ES, many overall goals might remain only in the mind of the designer. Indeed, some of the advantages of ES are in that one can decompose powerful systems into simple scenarios that are oblivious not only to each other's functions, as would be in standard object-oriented and structured programming encapsulation, but also to each other's very existence, since the various modules do not even call each other; they only exist and run in parallel. One can readily observe, however, that many of the original overall intentions and requirements, whether tacit or already documented, *can* be formalized in some way. Or, at least, one can automate the recognition *at development time* of whether certain goals are achieved or not, in observing actual system behavior in test runs.

Consider, for example, an at-home patient-assistant robot, or an autonomous service vehicle for factory yards, and focus on the task of fetching and delivering an object. Regardless of how each of these systems is programmed to do its tasks, one may be able to specify explicit criteria for recognizing, say, efficient vs. inefficient routes, steady vs. unsteady motion, successful vs. unsuccessful object or location recognition, safe vs. unsafe negotiation of obstacles, etc.

Further, in addition to, or instead of, such explicit specifications, one may program AI-based 'recognizers' for such properties. Since these are development-time recognizers, their input can be visual videos or trace/log files with actual data, like commands, environment signals, objects, object locations and speeds, etc. Further, neither the ES-based nor the AI-based recognizers have to complete their tasks in real time, which contributes to making them feasible.

Verifying Specifications Against System Goals

Executable system specifications mostly capture how the system should behave under various conditions and following certain events. They focus on relatively local behavior, and have only a limited awareness of overall system goals. During the classical formal verification of specifications, or automated program synthesis, one may introduce some formalization of overall desired results, at various abstraction levels. The actual formal verification of the system, or even that of only a single component, is, however, often intractable. Overall behavior is thus often verified only at high abstraction levels, and precise formal verification is achieved only for smaller components with well-defined boundaries.

Further, during development, formal methods may be able to check if given specifications are realizable, or if an implementation meets its specifications, but these methods might miss tacit needs, which are noticed only at much later stages.

A step towards addressing this limitation can be by using the goal-accomplishment recognizers discussed earlier. Thus, in addition to employing ordinary restricted test cases and limited formal verification, one may conduct extensive automated runs, with a variety of coverage-related parameters, and examine where the existing system or component indeed seems to meet its goals.

Succinct Description of Functional Gaps

When presented with a failed run or a counter-example resulting from formal verification, engineers can often generalize it to provide a broader description of gaps between what the system presently does and what it should do. E.g., in the above home-assistant robot and factory vehicle example, one may translate a near collision into a conclusion such as "the robot does not understand that certain obstacles, like a wet-floor sign or a horizontal area-demarcation tape, mark an entire forbidden area, which must not be entered , even when the path is physically open". Or, a single spilling of some drink may be translated into "the robot changes of speed and direction are such that they may cause open liquid containers like a glass of tea or a bucket of paint, to spill over". While engineers may offer such summaries of causalities and conditions after seeing only one example, automating this process can present a significant challenge, requiring multiple examples, and a-priori hints or constraints about the kinds of statements that would appear in the description (Alur et al. 2013).

Thus, regardless of the manual or automated methods used to create the gap description (not yet addressing the gap), when verification or testing shows that a well-defined goal is not met, our approach calls for creating a formal representation of the gap between the present capabilities of the system and those required by the goal. Such descriptions should emphasize the specific gap and its boundaries, comparable to sentences that a coach might use to draw an athlete's attention to a particular motion within an exercise, as opposed to just showing the entire exercise done correctly.

Gap descriptions can come in a variety of forms: a concise mathematical specification of a desired reaction, paired with a specification of how the system does it at present; a set of *multiple* concrete simulations of counterexamples, paired with desired runs, with the differences highlighted; in some cases the very component that closes the gap (as described in the next section) can serve as its definition (e.g., the detection and correct handling of previously-unrecognized obstacles), but this does not have to be the case. E.g., the correction of the spilling from liquid containers could come from slower and smoother robot motions, or from checking that liquid containers are closed well, or, alternatively, from checking that such containers are not filled to the top.

Closing Gaps with Dedicated Components

Given a concise representation of a particular gap between the capabilities of a given specification and a given goal, we propose that, rather than improving the existing components such that they also handle the gap area correctly, one should create one or more dedicated components (AI- or ES-based) that address the gap as their only, or their main, goal.

Going back to our example systems, for handling wet-floor signs or closed-off areas, a new component can detect these objects and feed its findings to existing components that would now handle the obstacle better. Alternatively, it can enforce smoother motions by either dictating new accelerations or angles of direction changes, overriding existing ones, or by constraining some non-deterministic acceleration and angle choices made by other modules. See the next section for more details about this composition.

Component Composition

We propose that applications be built in ways that allow incremental addition of components, where each component is relatively oblivious to the others, is responsible only for one or very few aspects of behavior, and is activated and composed with the rest of the system by the run-time infrastructure, rather than by invoking, or being invoked by, other application modules. Examples for such techniques (each with its capabilities and limitations) include aspect oriented programming (Kiczales et al. 1997), feature oriented software development (Apel and Kästner 2009), BIP (Behavior, Intention, Priority) (Bliudze and Sifakis 2008), and scenario-based programming (SBP) (a.k.a. behavioral programming) (Damm and Harel 2001; Harel and Marelly 2003; Harel, Marron, and Weiss 2012).

In particular, we observe that AI-based components can be composed smoothly with SBP, as follows. In SBP, system behavior is abstracted as events. Each component is a scenario that can wait for events (coming from the environment or the system), and then propose events that should be considered for triggering (termed *requested events*). The scenario can also declare events as forbidden (termed *blocked events*) until some other event occurs. All scenarios run in parallel, and an event-selection mechanism in the run-time infrastructure selects the next event and notifies and resumes all affected scenarios. All resumed scenarios then proceed to their next synchronization point and present their (possibly revised) declarations of requested and blocked events. The process then repeats. *Sensor scenarios* translate real environment events (e.g., receiving a voice command or noticing an object in the environment) into behavioral events, and *actuator scenarios* translate certain behavioral events into their respective physical changes in the environment (e.g., displaying some text or turning a vehicle's wheels).

While common SBP specification contain very crisp rules, like "when the traffic light is red the vehicle cannot move forward until the light is green", pure AI-based components can still readily fit into this framework in several ways: they can serve as sensor components, translating complex environment conditions, as captured, e.g., in a raw image of a street intersection, into concrete and succinct behavioral events, e.g., "the traffic light changed to green"; they can take over certain actuator functions, e.g., physically controlling motor speed changes; or, they may join the SBP scenarios in deciding on the next event, requesting and blocking certain events, e.g., if the changing of the motor speed allowed multiple non-deterministic choices, the AI component can block the choices that it deems to be undesired. This composition may, of course, be augmented with a variety of collaboration and joint decision techniques, like (Kaminka and Frenkel 2007; Bliudze and Sifakis 2008).

Transitioning AI components towards ES

While AI solutions using neural nets, search or the emergent behavior of collaborating agents are powerful, there is a constant pressure to replace them with explicit specification and even with conventional algorithmic code. Thus, as more knowledge is gained about the problem, one may wish to strengthen parts of the solution by applying specific procedural components; stakeholders and regulators may demand more detailed explanation of how and why a particular neural net works, and the explanation may be used to actually re-develop aspects of the solution; or, performance issues may emerge which slow down the AI-based solution, driving towards replacement with efficient, explicit specifications.

Such transitioning from AI to ES, however, does not have to be complete. Certain parts of the AI solution may be transformed into ES components while others are redeveloped using AI techniques, but addressing a 'smaller' problem, which is part of the original problem. (The word 'smaller' appears here in quotes, since often the issues at hand can be infinitely or unboundedly complex, preventing a meaningful definition and comparison of problem sizes.) For example, assume that the robot had an AI-based solution for obstacle detection and avoidance. This may be refined into having an ES-based handling of well-recognized obstacles, such as walls, fences, and standard no-entry traffic signs, combined with an AI-based handling of other kinds of obstacles, including the abovementioned wet-floor signs or tapes that close off an area. This may then be further refined, towards using AI only for *recognition* of the special obstacles, but having explicit ES rules for the reactive behavior of computing the boundaries of the restricted area.

Similarly, an AI-based solution for controlling a robot's speed and direction changes, may be divided into explicit ES rules for computing rotation and acceleration parameters depending on the nature of the payload, path conditions and urgency, while 'smaller' AI-based solutions may be used to detect 'only' the input to these rules, i.e., the payload properties, path conditions or scheduling constraints expected by the user (see, e.g.,(Rosenfeld and Kraus 2018)).

Converting AI to ES may be done manually, but it can also be assisted by automated tools. For example, specification-mining tools can observe actual behaviors of a system containing a neural net, in the form of trace/log files or input-output examples, and infer scenarios and event-sequence causalities (Ammons, Bodik, and Larus 2002; Lo and Maoz 2008), algorithms (Beschastnikh et al. 2013), or procedural programs (Balog et al. 2016) that carry out the related function or a subset thereof.

Related Work

Artificial Intelligence and software engineering have evolved as separate disciplines within computer science. In recent years, however, there are research efforts and calls for action, intended to bring about cross fertilization of techniques and approaches between the two (see, e.g., (Xie 2018)). These are manifested in a variety of forms, in development tools, testing, risk analysis, and of course in the running system itself. Specific examples include applying rigorous software engineering methods in developing collaborating agents (Jiménez, Piattini, and Vizcaíno 2009; Sturm and Shehory 2014), recent efforts to apply formal verification techniques to neural nets (Katz et al. 2017), using AI algorithms for speeding up automated program synthesis and formal verification in bounded model-checking (Biere et al. 2009), and using AI planning algorithms for run-time lookahead in formally-specified systems (Harel and Segall 2007; Weinstock 2015; Brukman et al. 2015). Additional examples can be found in (WISE Intl. Workshop 2017).

In this paper, we focused on a particular form of architectural integration: dividing a system into modules based on AI and modules based on executable specifications, and composing these two kinds of components via underlying run-time infrastructure services.

Discussion and Conclusion

Clearly, each of the principles of our approach to developing and designing the architecture of complex systems — automated detection of goal compliance, verification against goals, explicit statement of gaps, developing modules that exactly address particular gaps, transparent infrastructure composition of AI and ES modules, and decomposing AI modules into ES and 'smaller' AI modules — presents a challenge in its own right. E.g., the design of the ES aspects of a system will have to overcome some of the challenges faced by earlier attempts at developing expert systems; the translation of a single failure into a formal, yet more general, description of a functional gap, may initially turn out to be more complex than in our examples; or, the decomposition of an AI-based component into smaller sub goals may not readily result in smaller and simpler components. However, based on preliminary observations, we believe that with appropriate research work it will be possible to demonstrate both the feasibility and advantages of the approach. These advantages include:

Intuitiveness. The division of the problem and goals into stand-alone modules responsible for various aspects of the problem, with little or no knowledge of, and interaction with, each other, is aligned with how humans often describe the behaviors and capabilities of systems — in requirements documents, in user manuals, or informally to each other.

Robustness. While we cannot at this stage say that a particular system built in this way will be better than one built using other methods, the ability to incrementally add features and resolve deficiencies with highly-focused and intuitive modules, seems like a promising enabler of achieving stability under a variety of conditions.

Testing and verification. The combination of having mechanisms for automatically detecting whether goals and subgoals in a given test or production run are met, having executable-specification modules that are in themselves formally verifiable, and having AI-modules with very well-scoped functions, strengthens the developers' ability to ensure the safety and correct operation of the system.

Potential for run-time goal awareness. Integrating automated development-time detection of whether goals are achieved, into the run-time real-world execution of the system can further improve the system's robustness and its ability to cope with conditions that were not seen earlier, or whose handling was not prescribed in advance.

Acknowledgments.

This research was supported by grants from The German-Israeli Foundation for Scientific Research (GIF), The Minerva Foundation, The Israel Science Foundation (ISF), Koshland postdoctoral fellowship, NSF grants IIS-1527668, IIS-1830549, and NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering".

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M.; Raghothaman, M.; Seshia, S.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-Guided Synthesis. In *FMCAD*.

Ammons, G.; Bodik, R.; and Larus, J. 2002. Mining Specifications. *ACM Sigplan Notices* 37(1):4–16.

Apel, S., and Kästner, C. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8(5):49–84.

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Beschastnikh, I.; Brun, Y.; Abrahamson, J.; Ernst, M. D.; and Krishnamurthy, A. 2013. Unifying fsm-inference algorithms through declarative specification. In *Conference on Software Engineering*, 252–261. IEEE Press.

Biere, A.; Cimatti, A.; Clarke, E. M.; Strichman, O.; and Zhu, Y. 2009. Bounded model checking. *Handbook of satisfiability* 185:457–481.

Bliudze, S., and Sifakis, J. 2008. A Notion of Glue Expressiveness for Component-Based Systems. In *Proc. 19th Int. Conf. on Concurrency Theory (CONCUR)*, 508–522.

Brukman, O.; Dolev, S.; Weinstock, M.; and Weiss, G. 2015. Self-* programming: run-time parallel control search for reflection box. *Evolving Systems* 6(1):23–40.

Damm, W., and Harel, D. 2001. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design* 19(1):45–80.

Greenyer, J.; Gritzner, D.; Katz, G.; and Marron, A. 2016. Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *MoDELS*. CEUR.

Harel, D., and Marelly, R. 2003. *Come, Let's Play*. Springer.

Harel, D., and Pnueli, A. 1985. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, 477–498.

Harel, D., and Segall, I. 2007. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *TACAS*, 485–499.

Harel, D.; Kugler, H.; Marelly, R.; and Pnueli, A. 2002. Smart Play-Out of Behavioral Requirements. In *FMCAD*.

Harel, D.; Kantor, A.; Katz, G.; Marron, A.; Mizrahi, L.; and Weiss, G. 2013. On composing and proving correctness of reactive behavior. *EMSOFT*.

Harel, D.; Marron, A.; and Weiss, G. 2012. Behavioral Programming. *Comm. of the ACM* 55(7):90–100.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3):231–274.

Jiménez, M.; Piattini, M.; and Vizcaíno, A. 2009. Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering* 2009:3.

Kaminka, G. A., and Frenkel, I. 2007. Integration of coordination mechanisms in the bite multi-robot architecture. In *Robotics and Automation, 2007 IEEE International Conference on*, 2859–2866. IEEE.

Katz, G.; Barrett, C.; Dill, D.; Julian, K.; and Kochenderfer, M. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. *arXiv preprint arXiv:1702.01135*.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; and Irwin, J. 1997. Aspect-oriented programming. *ECOOP'97* 220–242.

Lo, D., and Maoz, S. 2008. Mining scenario-based triggers and effects. In *ASE*, 109–118. IEEE.

Rosenfeld, A., and Kraus, S. 2018. *Predicting Human Decision Making: From Prediction to Intelligent Agent Design*. Morgan & Claypool.

Sturm, A., and Shehory, O. 2014. Agent-oriented software engineering: revisiting the state of the art. In *Agent-Oriented Software Engineering*, 13–26. Springer.

Vardi, M. Y. 1995. An automata-theoretic approach to fair realizability and synthesis. In *International Conference on Computer Aided Verification*, 267–278. Springer.

Weinstock, O. M. 2015. Online search in behavioral programming models. *Proceedings of the ACM Student Research Competition at MODELS* 58–63.

WISE Intl. Workshop. 2017. WISE Intl. Workshop on Intelligent Software Engineering, co-located with ASE. URL: https://isofteng.github.io/wise2017/.

Xie, T. 2018. Intelligent Software Engineering. SETTA keynote, https://www.slideshare.net/taoxiease/setta18-keynote-intelligent-software-engineering-synergy-between-ai-and-software-engineering , accessed 9/2018.