Self-Explaining Digital Systems – Some Technical Steps

Goerschwin Fey[1] Rolf Drechsler[2,3]
[1]Hamburg University of Technology, 21071 Hamburg
[2]University of Bremen, 28359 Bremen
[3]DFKI, 28359 Bremen

Abstract

Today's increasingly complex adaptable and autonomous systems are hard to design and difficult to use. Partly this is due to problems in understanding why a system executes certain actions. We propose to extend digital systems such that they can explain their actions to users and designers. We formalize this as *self-explanation* and show how to implement and verify a self-explaining system. A robot controller serves as proof-of-concept for self-explanation.

1 Introduction

Digital systems continuously increase in their complexity due to integration of various new features. Systems handle failures and have complex decision mechanisms for adaptability and autonomy. Understanding why a system performs certain actions becomes more and more difficult for users. Also designers have to cope with the complexity while developing the system or parts of it. The main difficulties are the inaccessibility of the inner logic of the digital system or a lack in understanding all the details. An explanation for actions executed by a digital system unveils the reasons for these actions and, by this, can serve various purposes.

From the outside a user may be puzzled why a technical device performs a certain action, e.g., "why does the traffic light turn red?" In simple cases the user will know the reason, e.g., "a pedestrian pushed the button, so pedestrians get green light, cars get red light". In more complex cases, explanations for actions may not as easily be accessible. When the digital system that controls the larger technical device provides an explanation, the user can understand why something happens. This raises the user's confidence in the correct behavior. The explanation for actions required in this case must refer to external input to the system, e.g., through sensors, and to an abstraction of the internal state that is understandable for a user.

Also designers of digital systems can benefit from explanations. A typical design task is debugging where a designer has to find the reason for certain actions executed by a digital system. Depending on the current design task a designer may use the same explanations that help users. Additionally, more detailed explanations, e.g., justifying data exchange between functional units may be useful. Thus, debugging and development are supported by explanations giving simple access points for a designer justifying the system's execution paths. At design time a designer can use explanations to understand the relation between the specification and the implementation.

Correctness of the system is validated through explanations if these explanations provide an alternative view that justifies the actual output. For in-field operation explanations may even be exploited for monitoring as a side-check that validates the actual execution of the system to detect failures and unexpected usage. In particular, problems are detected earlier when explanations cannot be generated, are not well-formed, or are not consistent with respect to the actual behavior.

Given a digital system the question is how to provide an explanation for observable actions online. While on first sight this mainly concerns functional aspects also non-functional aspects like actual power consumption or response time of the system deserve explanations.

During online operation either the system itself or some dedicated additional entity must provide the explanations. This incurs a cost, e.g., for storing historical data that explains and, by this, also justifies current and future actions. This overhead must be kept as low as possible.

A non-trivial challenge is to provide concise explanations in a cost-efficient way. While some actions of a system may have very simple explanations, e.g., "the power-on button has been pressed", other actions may require a deep understanding of the system, e.g., "when the distance to an energy source is large and the battery level is low, we save energy by reducing light as well as speed and move towards the energy source". Such an explanation may in turn require knowledge about what an energy source is, what thresholds are used, and how the system detects where the next energy source may be found.

Our contributions are the following:

- We formalize explanations and define what a self-explaining system is. We explain how to verify whether a system is self-explaining.
- We provide a technical solution for explanations on the functional level and explain how to automatically infer and to extend explanations to non-functional aspects.
- We consider a robot controller implemented at the register transfer level in Verilog as a case study.

The paper is structured as follows: While there is no directly related work, Section 2 considers the aspect of explanation in other areas. Section 3 formalizes explanations, defines a self-explaining system, its implementation and verification. Section 4 studies a self-explaining controller for an autonomous robot and explains how explanations may automatically be inferred. Section 5 draws conclusions.

2 Related Work

The concept of self-explaining digital systems is new but related to explanation as understood in other domains. Thus, there is no tightly related work. But various communities are interested in a deeper understanding of systems, implementations, or algorithms for different reasons as discussed in the following.

Causation has a long history in philosophy where [15] is a more recent approach that relates events and their causes in chains such that one event can cause a next one. Often underlying hidden relations make this simple approach controversial. A rigorous mathematical approach instead can use statistical models to cope with non-understood as well as truly non-deterministic dependencies [26]. Artificial intelligence particularly in the form of artificial neural networks made a significant progress in the recent past modeling such relations. However, given an artificial neural network it is not understandable how it internally processes data, e.g., what kind of features from the data samples are used or extracted, how they are represented etc. First approaches to reconstruct this information in the input space have been proposed [9, 21].

Decision procedures are a class of very complex algorithms producing results needed to formally certify the integrity of systems. The pairing of complexity and certification stimulated the search for understanding the verdict provided by a decision procedure. Typically, this verdict either yields a feasible solution to some task, e.g., a satisfying assignment in case of Boolean satisfiability (SAT) solver, or denies the existence of any solution at all, e.g., unsatisfiability in case of SAT solving. A feasible solution can easily be checked. Understanding why some task cannot be solved is more difficult. Proofs [10, 30], unsatisfiable cores [25] or Craig-interpolants [11] provide natural explanations.

Understanding complex programs is a tedious task requiring tool support [29]. One example is the analysis of dataflow in programs and of root causes for certain output. Static [28] and dynamic [13] slicing show how specific data has been produced by a program. Dynamic dependency graphs track the behavior, e.g., to extract formal properties [19].

Debugging circuits is hard due to the lack of observability into a chip. Trace buffers provide an opportunity to record internal signals [5]. The careful selection of signals [18] and their processing allows to reconstruct longer traces. Coupling with software extensions allows to much more accurately pin point time windows for recording [16].

Verification requires a deep understanding of a system's functionality. Model checking is a well established and automated approach for formal verification. Typically, logic languages like *Linear Temporal Logic* (LTL), *Computation Tree Logic* (CTL), or *System Verilog Assertions* (SVA) are used to express properties that are then verified. These properties summarize the functionality of a design in a different way and thus explain the behavior. Verification methodology [2, 1] ensures that properties capture an abstraction rather than the technical details of an implementation.

Beyond pure design time verification is the idea of proof carrying code to allow for a simplified online verification before execution [24].

Self-awareness of computing systems [12] on various levels has been proposed as a concept to improve online adaption and optimization. Application areas range from hardware level to coordination of production processes, e.g., [23, 27]. The information extracted for self-awareness relates to explanation usually focused towards a specific optimization goal.

While all these aspects relate to explanation, *self-explanation* has been rarely discussed. For organic computing, self-explanation has been postulated as a useful concept for increasing acceptance by users [22]. The human-oriented aspect has intensively been studied in intelligent human computer interfaces and support systems [20]. Self-explanation has also been proposed for software systems although limited to the narrow domain of agent-based software [7] and mainly been studied in the form of ontologies for information retrieval [8]. Expert systems as one very relevant domain in artificial intelligence formalize and reason on knowledge within a specific context with the goal to diagnose, control, and/or explain. Particularly, real-time expert systems have been proposed, e.g., for fault tolerance [17]. Aspects like online-reasoning on formalized knowledge have been considered in this domain.

The overview in [6] introduced an abstract concept for self-explanation and also discusses in-field verification and security under reconfiguration. However, that paper lacks the technical details on self-explanation as it does neither provide a formalization nor an implementation approach nor a case-study.

This brief overview of very diverse works in several fields shows that understanding a system has a long tradition and is extremely important. Recent advances in autonomy and complexity reinforce this demand. In contrast to previous work, we show how to turn a given digital system into a self-explaining system.

3 Self-Explanation

Figure 1 gives a high-level view for self-explanation as proposed here. The digital system is enhanced by a layer for self-explanation that holds a – potentially abstracted – model of the system. Any action executed by the system at a certain point in time is an event (bold black arrows in the figure). The explanation layer stores events and their immediate causes as an explanation and provides a unique tag to the system (dotted black arrows). While processing data, the system relates follow-up events to previous ones based on these tags (blue zig-zag arrows). Besides events, references to the specification can provide causes for actions. The user or designer may retrieve an explanation for events observable at the output of the system as a cause effect chain (green dots connected by arrows). This cause effect chain only refers to input provided to the system, the – abstracted – system model, and the specification.

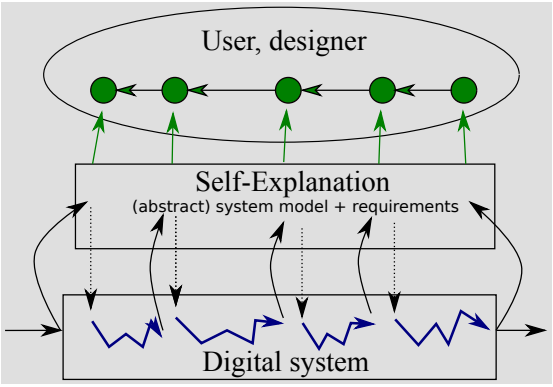

Figure 1 Approach

In the following we formalize self-explanation, provide an approach for implementation and verification. We also propose a conceptual framework that uses different layers for making explanations more digestible for designers and users, respectively.

3.1 Formalizing Explanations

We consider explanations in terms of cause-effect relationships. Before defining explanations we describe our system model. The system is represented by a set of variables V composed of disjoint sets of input variables I, output variables O, and internal variables. A variable is mapped to a value at any time while the system executes.

This system model is quite general. For a digital system a variable may correspond to a variable in software or to a signal in (digital) hardware. For a cyber-physical system a variable may also represent the activation of an actuator or a message sent over the network.

Based on this system model we introduce our notion of actions, events, causes, and explanations to formalize them afterwards. An *action* of a system fixes a subset of variables to certain values[1]. An *observable action* fixes observable output values of the system. An *input action* fixes input variables that are not controlled by the system, but by the environment. An action executed at a specific point in time by the running system is an *event*. We assume that a set of requirements is available for the system from a precise specification. A *cause* is an event or a requirement. An *explanation* for an event consists of one or more causes.

These terms now need more formal definitions to reason about explanations. An *action* assigns values to a subset of either I, O or $V/(O \cup I)$ of the variables introduced above. We define an ordered set of actions $\mathscr{A}$ with $i(a)$ for $a \in \mathscr{A}$ providing the unique index of a, a set of requirements $\mathscr{R}$ and the set of explanations $\mathscr{E} \subseteq \mathscr{A} \times \mathbb{N} \times 2^{\mathscr{R}} \times 2^{\mathscr{A}} \times \mathbb{N}^{|\mathscr{A}|}$. An *explanation* $e = (a,t,R,A,T) \in \mathscr{E}$ relates the action a with unique tag t, i.e., the event (a,t) to its causes. The tag t may be thought of as the value of a system-wide wall-clock time when executing the action. However, such a strong notion of timing is not mandatory. Having the same tag for a particular action occurring at most once is sufficient for our purpose and is easier to implement in an asynchronous distributed system. The vector T in an explanation relates all actions in A to their unique tags using the index function $i(a)$ such that $a \in A$ is related to the event $(a, T_{i(a)})$ where T_j denotes the jth element of vector T. Since $A \subseteq \mathscr{A}$ the relation $|A| \leq |T|$ holds, so unused tags in T are simply disregarded. Technically, the reference to prior events directly refers to their explanations. Note, that there may be multiple justifications for the same action, e.g., the light may be turned off because there is sufficient ambient light or because the battery is low. We require such ambiguities to be resolved during run time based on the actual implementation of the system.

Lewis [15] requires for counterfactual dependence of an event e on its cause c that $c \rightarrow e$ and $\neg c \rightarrow \neg e$. However, an event is identified with precisely the actual occurrence of this event. There may be alternative ways to cause a similar event, but the actual event e was precisely due to the cause c. Consider the event that "the window was broken by a stone thrown by Joe". The window may have alternatively been broken by a ball thrown by Joe, but this would have been a different "window broken" event. Lewis achieves this precision by associating the actual event e with a proposition $O(e)$ that is true iff e occurs and false otherwise. These propositions allow to abstract from the imprecise natural language. Here we achieve this precision by adding tags to actions.

Lewis [15] defines causation as a transitive relationship where the cause of an event is an event itself that has its own causes. Similarly, we go from an event to the causes and from these to their causes until reaching requirements or inputs of the system.

Definition 1 *For an explanation* $e = (a,t,R,A,T)$*, the* immediate set of explanations *is given by* $E(e) = \{e' = (a',t',R',A',T') \in \mathscr{E} | a' \in A \text{ and } t' = T_{i(a')}\}$.

Definition 2 *We define the* full set of explanations $E^*(e)$ *as the transitive extension of* $E(e)$ *with respect to the causing events, i.e.,*

if $e' = (a',t',R',A',T') \in E^*(e)$ *and*
there exists $e'' = (a'',t'',R'',A'',T'') \in \mathscr{E}$
with $a'' \in A'$ *and* $t'' = T'_{i(a'')}$,
then $e'' \in E^*(e)$.

Now we define well-formed explanations that provide a unique explanation for any action and must ultimately be explained by input data and requirements only:

Definition 3 A set of explanations $\mathscr{E}$ is well-formed, *iff*

1. *for any* $e = (a,t,R,A,T) \in \mathscr{E}$ *there does not exist* $e' = (a,t,R',A',T') \in E^*(e)$ *with* $(R,A,T) \neq (R',A',T')$,
2. *for any* $e \in \mathscr{E}$ *if* $e' = (a',t',R',A',T') \in E^*(e)$ *then for any* $a'' \in A'/A'_{\downarrow I}$ *where* $A'_{\downarrow I}$ *is the set of actions in* A' *that fix values of inputs* I *there exists* $(a'',t'',R'',A'',T'') \in E^*(e)$.

[1] An extension of our formalism could consider more complex actions that include a certain series of assignments over time, e.g., to first send an address and afterwards data over a communication channel. However, for simplicity we assume here that an appropriate abstraction layer is available. Nonetheless, multiple valuations of the variables may be associated to the same action, e.g., the action "moving towards front left" may abstract from the radius of the curve followed by a system

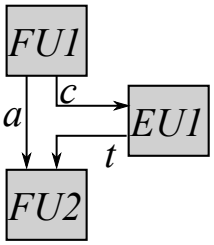

Figure 2 Implementation

Note, that our notation is similar to classical message-based events for formalizing asynchronous distributed systems, e.g., used in the seminal work of Lamport [14] that explains how to deduce a system-wide wall-clock time. An important difference is, however, that in our case the execution of the system perfectly describes the order of events. An explanation then captures this order without additional mechanisms for synchronization. The only requirement is the association of an action with a unique tag to form an event, i.e., any action occurs at most once with a particular tag.

Our formalism provides the basis for extending an actual digital system to provide explanations for each observable action. The sets of variables, actions, requirements, and explanations are freely defined. This leaves freedom to decide on the granularity of explanations available during run time, e.g., whether an action only captures the driving direction of a robot or the precise values of motor control signals.

The set of observable actions must be derived at first. The methodology must ensure that for each possible system output there is a related action.

Definition 4 A set of observable actions is complete *with respect to a given system iff for any observable output of the system there exists a related observable action.*

Definition 5 A set of explanations is complete *iff it is well-formed and explains all observable actions.*

Definition 6 *A* digital system is self-explaining *iff it has a complete set of observable actions and creates a complete set of explanations.*

3.2 Implementation

Practically, explanations are produced by adding appropriate statements to the design description. To create the cause-effect chain, we think of functional units that are connected to each other. A functional unit may be a hardware module or a software function. To produce explanations for the actions, each unit records the actions and their explanations from preceding units together with the input. By this, data being processed can always be related to its causes, likewise actions triggered by that data can be associated to their causes.

In the following we describe a concrete approach to implement a self-explaining digital system. Functional units derive causes for their actions. We associate an *explanation unit* for storage, reference, and usage of explanations to each functional unit. Whenever a functional unit executes an action, the cause for that action is forwarded to the explanation unit. The explanation unit then provides unique tags for the action to form an event, merges it with the cause, and stores the resulting explanation. Other functional units query the explanation unit to associate incoming data with an event and its explanation. This information must then be passed jointly while processing the data to provide the causes for an action. Figure 2 illustrates this. Functional unit *FU1* executes an action a passed to functional unit *FU2*. The cause c of a is stored in explanation unit *EU1* that provides a unique tag t. *FU2* refers to the event (a,t) to derive causes for its own actions.

For this step we rely on the designer to enhance the implementation with functionality to pass causes and drive explanation units by adding appropriate code. The designer also decides whether actions are defined in terms of existing variables of the design or whether new variables are introduced to allow for abstraction.

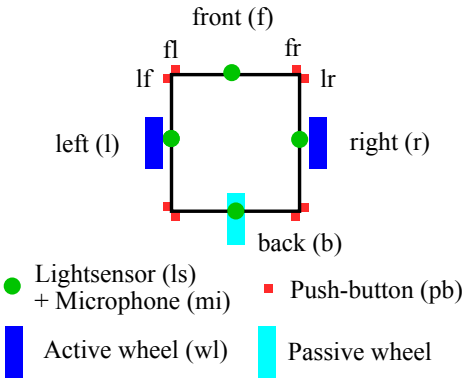

Figure 3 Robot

4 Case Study

We apply our approach to a small robot controller that explains actions executed with the controlled robot. Figure 3 shows an abstract view of the actual robot. The robot has wheels on the left and on the right side, each equipped with a motor that is commanded by the robot controller. The passive wheel on the back turns freely such that by commanding the two motors the robot controller can steer and move the robot. The main sensors of the robot are light-sensors and microphones on the four sides as well as eight push-buttons at its corners to detect collisions.

The specification in Table 1 describes the functionality. The robot controller moves the robot towards the noise detected by the microphones as long as the power levels indicated by the battery are sufficient. When power gets low, the controller steers towards the light detected by the light-sensors. Upon a collision detected by a push-button, the robot turns away from that button's contact point.

The four boxes shown in Figure 4 implement the robot controller in Verilog modules. Thus, in this case a functional unit directly corresponds to a Verilog module. Sensors and battery provide input data to the controller that provides output to the motors. The battery state directly impacts the motor speed.

Table 1 Specification

No.	content
R0	There are three battery levels: strong, medium, low
R1	If battery level is strong, move towards noise.
R2	Unless battery level is strong, move towards light.
R3	If battery level is low, use only half speed.
R4	If push-button is pressed, move towards other direction overriding requirements R0 to R3.

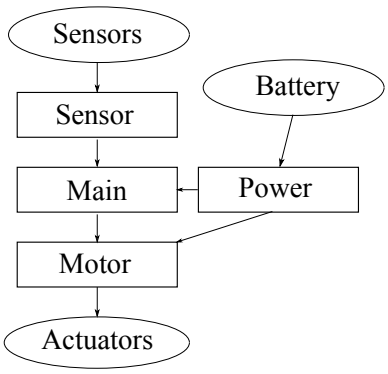

Figure 4 Modules of the robot controller

4.1 Adding Explanations

We consider cause-effect chains on the unit-level where actions fix the output values. Each module is equipped with an extra output that provides all causes for an action to form an explanation for each event. All kinds of actions are known in advance, so their dependence on causes is hard-coded into the system. Each module explains all its output data. The causes for the output data of one module are generated by preceding module's actions and requirements, so the causes explaining an action are encoded as a bit string for referencing them.

The explanations of the robot controller already make an abstraction from actual data. For example, instead of explaining the precise speed of the two motors, this is abstracted to one of the driving directions "straight", "forward left", "forward right", or "turn".

To have reproducible explanations and record their dependence, we equip every module with a separate explanation module. The explanation module stores explanations and provides unique tags for events. An explanation module essentially is a memory that stores the explanation which is a bit vector encoding the causes for an action. The memory address serves as the unique tag for the event associated to the current action of the respective module. By this, the unique tag also serves as reference to the explanation for the event. This tag is accessible for subsequent modules to produce their explanations. Uniqueness of these tags for events is subject to the limited size of the memory. By using a simple wrap-around arithmetic for the memory addresses, the size of the memory in the explanation module decides on the length of the history that can be recorded.

For example, the main module's explanations always depend on the actions of the sensor module and the power module together with the respective requirements. Receiving data from the power module or the sensor module corresponds to an action of these modules associated to an explanation with a unique tag. The main module stores the unique tags for the explanations to generate the explanation for its own action. This totals to 20 bits; in our implementation we used 24 bits to conveniently represent direction and sensors using hexadecimal digits. Explanations for the other modules have different lengths depending on their needs.

Table 2 Implementation sizes

Column "entries": number of addresses in explanation units
Column "#state bits" and "#gates": size of the implementation

	entries	#state bits	#gates
no explanation	-	113	5,692
with explanation	4	437	8,643
with explanation	32	2,250	21,714
with explanation	256	16,605	123,572

4.2 Results

Figure 5 shows an excerpt of the recorded explanations where nodes denote events and edges lead from a cause to an event. In this excerpt sensor input and powerstate ultimately explain driving direction and speed. Node "Main: 21" gives an explanation with unique tag "21" for the main module. According to the powerstate medium (node "Power: 02") and Requirement R2 the robot goes "straight" to the lightsensors "ls". This is one reason for the observable actions in nodes "Motor_left: 21" and "Motor_right: 21". The other reason is the current powerstate.

Figure 6 shows a similar explanation, but the powerstate changed from low to medium after deciding direction and speed in the main module and before adjusting the speed in the motor driver. Whether this is wanted or not depends on the implementation. Definitely, this explanation gives some insight into the behavior.

The original design has 257 lines of code, extensions for self-explanation require 119 lines, and the explanation unit has 28 lines. Table 2 gives an impression about the design and the cost for explanation. The numbers of state bits and gates are shown for four configurations: the plain robot controller without explanation and with explanation with sizes of 4, 32, and 256 entries in the memories of the explanation modules. In the table these memories are counted as state bits plus the decoding and encoding logic that adds to the gates in the circuit. For memories with 256 entries about 2 KByte of memory are required (the numbers in the table count bits). Note, that the encoding of explanations was not optimized for size. The main aims were a simplified implementation and easily separable reasons in a hexadecimal representation. A rather typical implementation of the controller would use microcontrollers connected over buses instead of pure Verilog. In that case a 2 KByte overhead for explanation would be rather small.

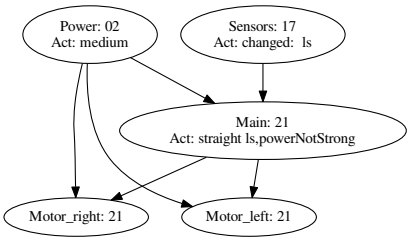

Figure 5 First excerpt from the explanations

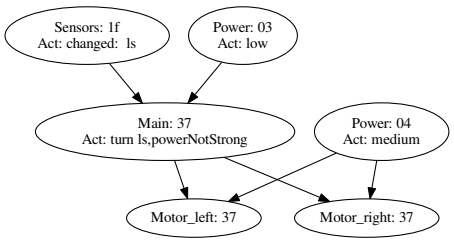

Figure 6 Second excerpt from the explanations

Table 3 Wrap around in tags for a trace of 10,000 cycles

Column "entries": number of addresses in explanation units
Other columns: number of wrap arounds for unique tags of modules

entries	Main	Motor_left	Motor_right	Power	Sensor
4	269	252	252	15	158
32	32	30	30	1	18
256	3	2	2	0	1

The number of entries in the memories decides for how long an explanation can be traced back before the unique tags for explanations wrap to zero again, i.e., are not unique anymore. This restricts the self-explanation to recent history. Table 3 shows how many times the tags were set back to zero for the different explanation units in a run of 10,000 cycles. The number of wrap arounds per module are different as the number of events also differs between the modules. Some of the events of one module do not necessarily trigger a follow-up event in a subsequent module, e.g., values of the microphones are only relevant, if the robot currently follows the sound. With 256 entries the length of the history increases to approximately 3,300 cycles for the main module having 3 wrap arounds.

Obviously, optimizations in the size required for explanations are possible, e.g., by adjusting the number of entries of explanation units per module or by encoding explanations in fewer bits. But this is not the scope of this paper which focuses on the concept of self-explanation.

4.3 Reasoning about Explanations

Having the design enhanced with explanations immediately supports a user or a designer in understanding the design's actions. Additionally, consistency of the explanations and the related actions is an interesting question. Due to the abstraction, e.g., in case of the driving direction it may not be fully clear what kind of actions precisely correspond to an explanation. We give some examples how to clarify this using model checking. We assume that the reader is familiar with model checking [4] so we do not provide any details for this process.

Considering the main module some facts can be analyzed by model checking, e.g., if the explanation of the main module says a certain action means moving "straight", this should imply that both motors are commanded to move in the same direction with the same speed. Indeed the simple robot controller always moves forward at full speed. In CTL this is proven using the formula:

$$AG(\ \text{exp}[23{:}20] = \text{straight} \rightarrow (\ \text{speed\_left}[7{:}0]=255 \wedge \text{speed\_right}[7{:}0]=255 \wedge \text{direction\_right=fwd} \wedge \text{direction\_left=fwd})\ )$$

The 24-bit vector "exp" refers to the explanation of the main module where only the bits corresponding to the description of the action are selected; the 8-bit vectors "speed_right" and "speed_left" correspond to the speed for the left and right motor, respectively; likewise the "direction"-variables.

Similar facts can be formalized for other situations. Using a more expressive language like SVA, properties may be formulated in an even nicer way, e.g., using expressions over bit-vectors. The underlying concepts for explanation remain the same.

4.4 Extensions

Currently, an action is defined to be a variable assignment. In practice, more complex actions may be of interest, e.g., to perform a burst access to a communication resource. Appropriate extensions are possible by allowing for a more general specification of an action, e.g., in terms of a formal property language that describes conditional sequential traces.

We propose completeness and well-formedness as basic criteria for self-explanation. Further properties of interest are aspects like determinism or consistency with an environment model. The systems considered here are limited to generating explanations for themselves and out of the available view onto the environment which is largely unknown to the system. If the system itself incorporates a more detailed model of the environment, the expected impact on the environment can also be incorporated into the explanations. This provides an even deeper insight for the observer of the system and would immediately allow to judge the consistency of explanations with the actual behavior. Potentially this serves as the basis for an autonomous diagnosis loop.

Non-functional aspects like reaction time or power consumption similarly require self-reflexive functionality in the system, e.g., to determine the current processing load or current sensors and a prediction on future activities. This again can be seen as a model of the environment within the digital system.

4.5 Automated Inference

Manually enhancing a design for self-explanation may be time consuming. Thus, further automation is useful. Technically, one option to automatically derive explanations is the use of model checking engines. Given a precise specification of an observable action in terms of a formal language, model checking can derive all possible ways to execute this observable action. Logic queries [3] may serve as a monolithic natural tool to identify causes. Deriving these causes in terms of inputs of a functional unit and then continuing to preceding functional units allows to automatically derive well-formed explanations. Completeness must be ensured by formalizing all observable actions properly.

5 Conclusions

Future complex systems driving real-world processes must be self-explaining. Naturally, our proposal is just one technical solution that cannot consider many of the alternative ways to create a self-explaining system.

Our paper provides a formal notion of self-explanation and a proof-of-concept realization. We studied a robot controller as a use case. We gave an idea on how to automatically provide self-explanations. The extension to reactive systems in general and to systems where new actions may be defined on-the-fly remains for future work.

6 Literature

[1] P. Basu, S. Das, A. Banerjee, P. Dasgupta, P.P. Chakrabarti, C.R. Mohan, L. Fix, and R. Armoni. Design-intent coverage: A new paradigm for formal property verification. *IEEE Trans. on CAD*, 25(10):1922 –1934, 2006.

[2] Jörg Bormann. *Complete Functional Verification*. PhD thesis, University of Kaiserslautern, 2009. English translation 2017.

[3] William Chan. Temporal-logic queries. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463, 2000.

[4] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT press, 01 2001.

[5] Sergej Deutsch and Krishnendu Chakrabarty. Massive signal tracing using on-chip dram for in-system silicon debug. In *Int'l Test Conf.*, pages 1–10, 2014.

[6] Rolf Drechsler, Christoph Lüth, Görschwin Fey, and Tim Güneysu. Towards self-explaining digital systems: A design methodology for the next generation. In *International Verification and Security Workshop (IVSW)*, pages 1–6, 2018.

[7] Johannes Fähndrich, Sebastian Ahrndt, and Sahin Albayrak. Towards self-explaining agents. In *Trends in Practical Applications of Agents and Multiagent Systems*, pages 147–154, 2013.

[8] Johannes Fähndrich, Sebastian Ahrndt, and Sahin Albayrak. Self-explanation through semantic annotation: A survey. In *Position Papers of the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2015.

[9] Raphael Féraud and Fabrice Clérot. A methodology to explain neural network classification. *Neural Networks*, 15(2):237 – 246, 2002.

[10] Eugene Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe*, pages 886–891, 2003.

[11] B. Keng and A. Veneris. Scaling VLSI design debugging with interpolation. In *Int'l Conf. on Formal Methods in CAD*, pages 144–151, 2009.

[12] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[13] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] David Lewis. Causation. *Journal of Philosophy*, 70(17):556–567, 1973.

[16] David Lin, Eshan Singh, Clark Barrett, and Subhasish Mitra. A structured approach to post-silicon validation and debug using symbolic quick error detection. In *Int'l Test Conf.*, pages 1–10, 2015.

[17] Wei Liu. Real-time fault-tolerant control systems. In Cornelius T. Leondes, editor, *Expert Systems*, pages 267–304. Academic Press, 2002.

[18] Xiao Liu and Qiang Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Design, Automation and Test in Europe*, pages 1338–1343, 2009.

[19] Jan Malburg, Tino Flenker, and Goerschwin Fey. Property mining using dynamic dependency graphs. In *ASP Design Automation Conf.*, pages 244–250, 2017.

[20] Mark T. Maybury and Wolfgang Wahlster, editors. *Readings in Intelligent User Interfaces*. Morgan Kaufmann Publishers Inc., 1998.

[21] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1 – 15, 2018.

[22] Christian Müller-Schloer and Sven Tomforde. *Organic Computing – Technical Systems for Survival in the Real World*. Birkhäuser, 2017.

[23] Mischa Möstl, Johannes Schlatow, Rolf Ernst, Henry Hoffmann, Arif Merchant, and Alexander Shraer. Self-aware systems for the internet-of-things. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–9, 2016.

[24] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91. Springer Berlin Heidelberg, 1998.

[25] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conf.*, pages 518–523, 2004.

[26] Judea Pearl. *Causality*. Cambridge University Press, 2010.

[27] Lydia C. Siafara, Hedyeh A. Kholerdi, Aleksey Bratukhin, Nima Taherinejad, and Axel Jantsch. SAMBA - an architecture for adaptive cognitive control of distributed cyber-physical production systems based on its self-awareness. *Elektrotechnik und Informationstechnik*, 135(3):270–277, 2018.

[28] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, 1981.

[29] Steven Woods and Qiang Yang. The program understanding problem: analysis and a heuristic approach. In *International Conference on Software Engineering*, pages 6–15, 1996.

[30] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, pages 880–885, 2003.