The Formal Specification of a Kitchen Environment

Sureyya Tarkan*
Human-Computer Interaction Lab & Department of Computer Science
University of Maryland
sureyya@cs.umd.edu

ABSTRACT

Programming allows children to acquire problem-solving skills that will be useful to them throughout their life. Tools that have successfully attracted children to programming in the past are now outdated because children's interests have evolved with the technology. It has become more important to provide real-world experience to improve children's ability in solving real problems. As our previous studies with children suggest, cooking is appealing to both girls and boys. However, the challenge is to come up with a formal specification of cooking that is precise and complete, but still is an age-appropriate and fun activity for children. Therefore, this paper attempts to formalize an event-based cooking language using formal specification methods.

1. INTRODUCTION

Programming allows children to acquire problem-solving skills that will be useful to them throughout their life. With this objective in mind, there are many tools developed to teach programming in the early ages of childhood [14, 15, 19]. These languages were based on the idea of moving an object character, which is capable of responding to certain commands, in a finite spatial environment. Via gathering sensor data from the environment, the object could only do very simple actions on the intersection points of some coordinate system to accomplish a predefined goal. Children had to program this character with a limited set of functions by taking into account the unwanted states and the constraints on the system. Later, newer versions of these languages were released to mitigate the effects of changing programming paradigms (e.g. object-oriented, event-driven languages).

Although all of these tools have been successful in terms of attracting children to programming in the past, they have now become outdated since children's interests also evolved with the technology. Nowadays, children like to have more hi-tech toys that are capable of doing more complicated things than simple moving, drawing, or sounds [1, 2] and programming should allow children to explore these scenarios as well. Therefore, novel tools have appeared [6, 12] that extended the idea and enabled children to move into the 3D world. However, the theme have remained the same – that there is a character who moves with given commands in some environment. The basic set of actions and their arguments were still similar to their older counterparts and because usually such instructions only include numbers and geometric terms, their capabilities were limited to teaching mathematical concepts rather than challenging children to solve a real-world problem.

Our previous studies with children showed that cooking is appealing to both girls and boys when provided with an interesting recipe to prepare [21]. In these studies, children were able to come up with the correct recipe and enjoy their preparations in a collaborative programming environment. As opposed to previous approaches, in real-life, children cannot cook without adult guidance and a programming tool can levitate their creativity by giving them such an opportunity. Advantages of using cooking to teach programming are: (i) recipes have an inherent structure similar to programs and concepts that are available in programming languages (e.g. looping, branching), (ii) cooking necessitates an understanding of mathematical concepts such as measurements, (iii) directions impose constraints in terms of time and order, (iv) recipes include objects (e.g. ingredients, utensils), (v) it is possible to parallel program multiple cooks, (vi) cooking is obviously a real-life experience.

Cooking is appealing to children [10] but we believe the following are some reasons why it was not used in the literature to teach programming. First of all, the core directions in cooking are far more complex in nature than simple moving instructions. Each direction's pre- and post-conditions have to take into account various situations not necessarily sensed with a single action. Thus, it is a non-trivial task to simplify the constraints of the environment to an age-appropriate level as it is with a coordinate system. Second, the terminology needs to be very precise. For example, here is an example from a children's cookbook [13]:

- **Blend:** to mix foods together until smooth.
- **Stir:** to continuously mix food with spoon.
- **Mix:** to stir two or more ingredients together until they are evenly combined.

The problem here is that *blend* and *stir* are not easy to differentiate; stirring looks like a special type of blending action since it uses a specific tool, however, looking at the outcomes of these actions, blending can be considered as a subclass of stirring actions because there are cases in which the food is left without being smooth enough. Apart from that, *mix* and *stir* are defined in terms of each other and this is ambiguous. Rather than raising children's understanding and problem-solving skills, such ambiguities may confuse children in the

*Supervised by Vibha Sazawal

Submitted to the Department of Computer Science, University of Maryland, College Park, MD 20742, USA in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

long run. Finally, closely related to the terminology issue is that there are various ambiguous directions in such recipes [11]. Consider the following instance, which in each step refers to one or some of the previous steps but there is no predefined structure:

1. sprinkle flour on board
2. roll dough
3. put on baking tray
4. add 1 tablespoon tomato sauce
5. spread on dough
6. add 3 zucchini slices
7. add 3 mushroom slices
8. sprinkle cheeses
9. bake 20 minutes
10. EAT

For example, there is an implicit assumption that the reader understands that the dough will be rolled on the floured board, the rolled dough will be put on the baking tray, etc. This is not easy to parse for a computer program and makes it difficult to follow the arguments and the return values of functions after a certain number of steps, like in the following sequence: on the spread dough add zucchini, to the dough with zucchini add mushroom, sprinkle cheeses to the dough with mushroom and finally, bake the cheese-dough. Because the inputs and outputs are not clear from the recipe, it looks as if it can be reordered without any problem, however, a recipe is strictly order-dependent. Moreover, some instructions have overloaded definitions, e.g. *sprinkle* can be called with or without a board that is not immediately apparent to the reader. The propositions are used interchangeably in some function calls above as *on* is used both before an ingredient and a tool. The usage of ingredients and measurements are not consistent either; note the use of *1 tablespoon* (natural number & measure together) and *3* (single natural number).

As these disadvantages suggest, the challenge is to come up with a formal specification of cooking that is precise and complete but still is an age-appropriate and fun activity for children. In this paper, we attempt to formalize an event-based cooking language using formal specification methods. Our contributions are twofold:

1. We provide an abstract Z [20, 23] specification of a kitchen environment. This specification is used to represent the state of the world and the operations defined on it.
2. We then, attempt to ensure that some constraints on this system are not violated. For that, we convert our Z specification into a model that can be put into the Alloy Analyzer [9] for fully automatic analysis.

The rest of the paper is organized as follows. First, we present previously designed languages in detail. Second, we show our full Z specification of the kitchen environment. Third, we explain our Alloy model definition. Then, we discuss the results and implications of our design. Finally, we conclude with future work.

2. RELATED WORK

Since language design for novices is a widely studied topic, there are many such examples in the history. One approach is to design a small and simple language called mini-language to teach basic programming. A common characteristic of such languages is that the user controls an actor in a micro-world.

The most well-known example of a mini-language that is accessible to children is the Logo programming language [14]. The Logo turtle is a simulated actor in a two-dimensional graphical world. Logo introduces programming through making the turtle draw simple pictures; for example, "forward 10," makes the turtle move in its forward direction. Starlogo [17], Leogo [5], and MultiLogo [16] are some versions based on the original Logo to introduce different paradigms in programming. Karel the Robot [15] is another most widely-used mini-language for beginners. Karel is a robot that lives in a simple grid-world that has streets in east-west, and avenues in north-south directions. There are also immovable walls and beepers. Karel can *move*, *turn*, *turn* itself *off*, and *sense* nearby walls and beepers. Karel++, Karel J Robot, J. Karel are variants of this mini-language that are used as introductions to different programming languages. Yet another one is the Jeroo [19] mini-language for introducing object-oriented programming. Jeroo is a rare mammal whose primary food source is the large Winsum flower. The Jeroos hop about the Santong island to pick and plant flowers. At the same time, they must avoid the water and evade or disable nets set by hunters. There are many other mini-languages that intend to teach programming to novices [3, 22]. Although our approach can be classified as a mini-language, we model a real-world scenario which is more complex than a simple grid micro-world. In comparison, the control commands require a deeper understanding than a micro-world command. The actors themselves are humans rather than robots or animals. There is no explicit mission to accomplish as in these scenarios and the end result (the dish) can be appreciated by everyone. We therefore, try to bring more realism to enhance novices' problem-solving skills.

As these shortcomings in modeling real-world phenomenon were observed by the language designers, they integrated new approaches to the design. Alice [6] and Scratch [12] are two recent and popular tools such that the user is given more freedom in terms of designing their models. Alice supports building 3D virtual worlds like short animated movies or games while Scratch makes it easy to create interactive stories, animations, games, music, and art. The features of the aforementioned mini-languages remain unaffected, i.e. the same mathematical and geometric concepts are still employed, but modeling has become more important. A major drawback is that the focus of children shifts from the primary goal of learning programming to designing better models and sprites when they get involved with the details of such a full-featured programming environment. Flexibility is important for extensibility, however, we are interested in teaching programming to kids. Thus, we take the approach of designing a novel mini-language that is based on a metaphor that is closely related to a real-world scenario with a target novice audience in a wider age range including younger children.

There are some cooking systems available for children. One of them is the Cooking Mama [7] game developed for the Nintendo. In particular, it is a cooking simulation in which dishes are prepared by completing at least two short mini-games, representing steps in the meal preparation process. Players use the Wii Remote to mimic real-life cooking movements. The player's performance is scored based on how quickly and accurately tasks are performed. The

game supports both single-, multi-player modes. However, the reviewers comment that cooking actions are difficult to perform successfully with the Remote control, leading to frustration [8]. Moreover, the game is not designed to teach programming to kids but rather to entertain them while our goal is specifically to teach programming.

3. Z SPECIFICATION

This section presents our formal language specification of the Kitchen Environment in Z notation that was type-checked using the Z/Eves tool [18]. We preferred to use an abstract definition, which does not define the necessary data structures, rather than an implementable concrete design.

We decided that an event-based programming language is a better fit to our task. Therefore, distinct from the Kitchen, the system has a Timer mechanism to keep track of which event to throw, at what time, and how to handle it. Kitchen's state is based on the current time. The pre-conditions and post-conditions of each schema, and how the state of the world evolves are provided within these declarations. If the users of the system call the operations appropriately, unexpected situations cannot happen. Error reports are produced for those situations when the preconditions are not met.

3.1 Type Declarations

We start formalizing our system by defining the basic types of the specification. `CNAME` represents a unique cook name while `INAME` is a unique ingredient name.

$$[CNAME, INAME]$$

Below are some values that other types can take in the system. `ENAME` stands for an event name, `MNAME` is a measurement name, `KNAME` is the general name of utensils, appliances, and tools that may be used, `REPORT` is defined for error handling.

$$\begin{aligned}
ENAME ::= {} & bakeDone \mid cleanDone \mid cookDone \mid \\
& cutDone \mid kneadDone \mid mixDone \mid \\
& preheatDone \mid putDone \\
MNAME ::= {} & teaspoon \mid tablespoon \mid cup \mid pint \mid quart \mid \\
& ounce \mid pound \mid package \mid pinch \mid gram \mid \\
& gallon \mid liter \mid hour \mid minute \mid celsius \mid \\
& fahrenheit \mid integer \\
KNAME ::= {} & counter \mid faucet \mid oven \mid refrigerator \mid table \\
REPORT ::= {} & ok \mid already_known \mid not_known \mid \\
& wrong_direction
\end{aligned}$$

Some type declarations use these primitive types. `KITCHEN_ITEM` has a name and a time value in which it announces that it is done. `AUTO` kitchen items work without manual control. We represent the `Cook` schema as a type (see Section 5). `Cook`s are referred by a unique name; has fields for time to start and end the action. `EID` is used to declare a unique identifier for each event. A `MEASUREMENT` is defined by a name and amount. `Ingredient` is another schema as type. Every `Ingredient` consists of a name, measure, and a set of other ingredients that changes with time. The types of ingredients are restricted to `BAKED`, `CUT`, `KNEADED`, `MIXED`, `PROCESSED`, and `RAW`. `Direction` stores the actual event calls for later use. `RECIPE` is a direction that is identified by the `EID` along with the items, measurements, and ingredients.

$$\begin{aligned}
& KITCHEN_ITEM == KNAME \times \mathbb{N} \\
& AUTO_ITEM == KITCHEN_ITEM \\
& Cook \mathrel{\widehat=} [cname : CNAME;\ initAction : \mathbb{N};\ endAction : \mathbb{N}] \\
& EID == ENAME \times \mathbb{N} \\
& MEASUREMENT == MNAME \times \mathbb{N} \\
& Ingredient \mathrel{\widehat=} [iname : INAME;\ measure : \\
& \quad MEASUREMENT;\ composedOf : \mathbb{N} \rightarrow \mathbb{P}\, INAME] \\
& BAKED_INGREDIENT == Ingredient \\
& CUT_INGREDIENT == Ingredient \\
& KNEADED_INGREDIENT == Ingredient \\
& MIXED_INGREDIENT == Ingredient \\
& PROCESSED_INGREDIENT == Ingredient \\
& RAW_INGREDIENT == Ingredient \\
& DIRECTION \mathrel{\widehat=} [c : Cook;\ item : KITCHEN_ITEM; \\
& \quad app : KITCHEN_ITEM;\ ingr : \mathbb{P}\, Ingredient] \\
& RECIPE == EID \rightarrow\ KITCHEN_ITEM \times \\
& \qquad KITCHEN_ITEM \times MEASUREMENT \times \\
& \qquad MEASUREMENT \times \mathbb{P}\, Ingredient
\end{aligned}$$

When the specification successfully executes, one would like to produce a report that indicates successful completion as shown with `Success` schema.

Success

$$result! : REPORT$$

$$result! = ok$$

3.2 State Specification

`Kitchen` schema acts as a database for our cooking language. There are `cooks`, `items`, `ingredients` in this world. It keeps track of `Available` cooks, items, and ingredients, `Dirty` and `Heated` items as well as `UsedIngredient`s. All of these entities depend on time (notice the relation symbols associated with each of them). `Recipe`, on the other hand, is used for checking if the order and arguments of events really comply with the actual directions that are specified within the given recipe.

Kitchen

$$\begin{aligned}
& cooks : \mathbb{P}\, Cook \\
& items : \mathbb{P}\, KITCHEN_ITEM \\
& ingredients : \mathbb{P}\, Ingredient \\
& AvailableCook : Cook \leftrightarrow \mathbb{N} \\
& AvailableItem : KITCHEN_ITEM \leftrightarrow \mathbb{N} \\
& DirtyItem : KITCHEN_ITEM \leftrightarrow \mathbb{N} \\
& HeatedItem : KITCHEN_ITEM \leftrightarrow \mathbb{N} \\
& AvailableIngredient : Ingredient \leftrightarrow \mathbb{N} \\
& UsedIngredient : Ingredient \leftrightarrow \mathbb{N} \\
& Recipe : RECIPE
\end{aligned}$$

$$\begin{aligned}
& \mathrm{dom}\ AvailableCook \subseteq cooks \\
& \mathrm{dom}\ AvailableItem \subseteq items \\
& \mathrm{dom}\ DirtyItem \subseteq items \\
& \mathrm{dom}\ HeatedItem \subseteq items \\
& \mathrm{dom}\ AvailableIngredient \subseteq ingredients \\
& \mathrm{dom}\ UsedIngredient \subseteq ingredients \\
& \forall t : \mathbb{N} \bullet \mathrm{dom}\,(AvailableItem \rhd \{t\}) \\
& \qquad \cap\, \mathrm{dom}\,(DirtyItem \rhd \{t\}) = \emptyset \\
& \forall t : \mathbb{N} \bullet \mathrm{dom}\,(AvailableIngredient \rhd \{t\}) \\
& \qquad \cap\, \mathrm{dom}\,(UsedIngredient \rhd \{t\}) = \emptyset
\end{aligned}$$

The predicates in the `Kitchen` schema indicate that `AvailableCook` is defined for those `cooks` in the `Kitchen`. Similarly, `AvailableItem`, `DirtyItem`, and `HeatedItem` have `items`

as their domain. `AvailableIngredient` and `UsedIngredient` are also in the domain of `ingredients`. There are two more important facts imposed on the system: that an item cannot be both `Available` and `Dirty` at the same time and analogously, `AvailableIngredient` and `UsedIngredient` are disjoint sets at all times.

`Timer` is a mechanism that updates the current time (`currTime`) and is distinct from the `Kitchen`. For every event that was added to the handler, `directions` keeps track of the properties (which cook was involved, what items and ingredients were used, etc.) of that event call in order later to properly associate them with the `Done` call. `currEvents` is a set of events that need to be handled at the `currTime`. `EventCount` counts the number of events and is practical in cases when multiple of the same type events are announced simultaneously (it gives them unique identifiers). The `EventHandler` associates events with the time that they are going to be announced. The predicates of this schema state that `directions` contain those event identifiers that were added to the `EventHandler` at some point. `currEvents` are a subset of those events in the `directions`. Additionally, at every point `EventHandler` contains events that have not been announced yet.

Timer

$$
\begin{array}{l}
currTime : \mathbb{N} \\
directions : EID \rightarrow Direction \\
currEvents : \mathbb{P}\, EID \\
EventCount : \mathbb{N} \\
EventHandler : EID \rightarrow \mathbb{N} \\
\hline
\mathrm{dom}\ EventHandler \subseteq \mathrm{dom}\ directions \\
currEvents \subseteq \mathrm{dom}\ directions \\
\forall\, e : EID \bullet currTime \leq EventHandler\,(e)
\end{array}
$$

The initial state of the `Kitchen` – dependent on the `Timer` – starts with given `cook?`s, and a default set of items: `counter`, `faucet`, `refrigerator`, `oven`, and `table`. Other sets are empty except that `Recipe` has to be non-empty. There are two crucial facts in the initial state. First, all ingredients are `RAW_INGREDIENT`s. Second, they are not compositions of other ingredients yet.

InitKitchen

$$
\begin{array}{l}
Kitchen \\
Timer \\
cook? : \mathbb{P}\, Cook \\
\hline
\mathrm{dom}\,(AvailableCook \rhd \{currTime\}) = cook? \\
\forall\, i : Ingredient \bullet \\
\quad i \in \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \\
\quad \Rightarrow i \in RAW\_INGREDIENT \\
\forall\, i : Ingredient \bullet i \in ingredients \\
\quad \Rightarrow \mathrm{ran}\,(\{currTime\} \lhd i\,.composedOf) = \emptyset \\
DirtyItem = \emptyset \\
HeatedItem = \emptyset \\
UsedIngredient = \emptyset \\
Recipe \neq \emptyset
\end{array}
$$

Aside from the `Kitchen`, the `Timer` initializes itself as follows. The `currTime` starts from the 0^{th} minute and `EventCount` is 0. `directions` and `currEvents` both start with empty sets.

InitTimer

$$
\begin{array}{l}
Timer \\
\hline
currTime = 0 \\
directions = \emptyset \\
currEvents = \emptyset \\
EventCount = 0
\end{array}
$$

3.3 Error Handling

Since we want to report errors that break the constraints of the system, below are some schemas that report errors based on whether the cook, or the kitchen item, or the ingredients are not defined. They all report `not_known` to the user. Note also that they do not change the state but simply quit the application with an appropriate message.

UnknownCook

$$
\begin{array}{l}
\Xi Kitchen \\
preparer? : Cook \\
result! : REPORT \\
\hline
preparer? \notin cooks \\
result! = not\_known
\end{array}
$$

UnknownTool

$$
\begin{array}{l}
\Xi Kitchen \\
tool? : KITCHEN\_ITEM \\
result! : REPORT \\
\hline
tool? \notin items \\
result! = not\_known
\end{array}
$$

UnknownIngredient

$$
\begin{array}{l}
\Xi Kitchen \\
what? : \mathbb{P}\, Ingredient \\
result! : REPORT \\
\hline
what? \not\subseteq ingredients \\
result! = not\_known
\end{array}
$$

Because cooking is an ordered activity and directions need to be strictly followed, we store the set of necessary directions in the `Recipe`. At each event call, the event's name and inputs are checked against that of the instructions from the `Recipe`, for tools, measurements, and ingredients. When there is a mismatch, the `WrongDirection` schema complains with an error message.

WrongDirection

$$
\begin{array}{l}
\Xi Kitchen \\
event? : EID \\
tool? : KITCHEN\_ITEM \\
where? : KITCHEN\_ITEM \\
duration? : MEASUREMENT \\
amount? : MEASUREMENT \\
what? : \mathbb{P}\, Ingredient \\
result! : REPORT \\
\hline
ran(\{event?\} \lhd Recipe) \neq \\
\quad \{(tool?, where?, duration?, amount?, what?)\} \\
result! = wrong\_direction
\end{array}
$$

3.4 Event Handling

Now that we have defined the basics of our system, we explain the dynamics of it using schemas to represent the operations. In the rest of this section, we elaborate on how the kitchen world evolves.

`Tick` is the `Timer` advancement mechanism which does not have a precondition but increases the `currTime` by one minute. It selects from the `EventHandler` all events, which are stored in the format (event-to-throw, minute-since-start), associated with that particular moment in time and passes them to `currEvents` so that they can be handled. Neither `directions` nor `EventCount` changes.

Tick

$$\Delta Timer$$

$$currTime' = currTime + 1$$
$$directions' = directions$$
$$currEvents' = currEvents \cup \mathrm{dom}(EventHandler \rhd \{currTime\})$$
$$EventCount' = EventCount$$
$$EventHandler' = EventHandler \rhd \{currTime\}$$

Events get recorded at the time of an event call if the current state of the world satisfies their pre-conditions. The handling of an event means associating the event with its matched `Done` event at the correct time to update the state according to the post-conditions. Therefore, there are four steps in the lifetime of an event: (i) The cook goes and picks up/turns on the corresponding kitchen item (see Figure 1 and schemas that use it), (ii) The tool works on completing the recipe direction (the time during which it is stored in the `EventHandler`[1]), (iii) The cook returns back to the table where it was working (see `CookDone`), (iv) The item finishes its job and announces its completion (see `EventDone` and schemas that use it). Note here that steps (iii) and (iv) may be done in parallel especially if the tool can work autonomously, otherwise, the cook has to be actively engaged in the successful completion of the operation. None of these steps are instantaneous and span some duration specified by the user or imposed by the system.

3.5 Event Specifications

Throughout this paper, we refer to `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` as events and an event completion as a `Done` event. All events can potentially change the state of the `Kitchen` database. The cook and the tool determine their own destinies with separate `Done` calls.

`Event` schema (Figure 1) uses the `Kitchen` during a change in the `Timer` state. This schema represents the common features in every event and thus, is included in all of the seven event calls. The pre-conditions are that the `preparer?` has to be an `AvailableCook` at the time of the call. The `duration?` specified by the user is in terms of `minute`s and `hour`s and has a non-zero value. The post-conditions are as follows. At the end of this event call, the cook becomes un`Available`. The number of events (`EventCount`) increases. Two new `directions` are created for `CookDone` and `EventDone` whose times are calculated with respect to the input `duration?` and the `tool?`'s type. These are simultaneously added to the `EventHandler`. There is no modification on the `currEvents` since it is the `Tick` event that determines which events are announced at the current time. This event spans the time between the initial call and the cook's response time (see `currTime` update in Figure 1). No ingredients' compositions are affected.

`Done` is another generic schema defined for done events. Similar to `Event`, it acts upon the `Kitchen` given the `Timer`. There is no change in `UsedIngredient`s, `EventCount`, or `directions` during this call. The given event identifier is used to remove this event from `currEvents`. `ename` is helpful in deciding which done event should be called since there are eight of them, i.e. `CookDone`, `BakeDone`, `CleanDone`, `CutDone`, `KneadDone`, `MixDone`, `PreheatDone`, `PutDone`.

Done

$$Kitchen$$
$$\Delta Timer$$
$$ename : ENAME$$
$$eid : EID$$

$$\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\})$$
$$EventCount' = EventCount$$
$$directions' = directions$$
$$EventHandler' = EventHandler$$
$$first\ eid = ename \land eid \in currEvents$$
$$currEvents' = currEvents \setminus \{eid\}$$

`EventDone` imports `Done` schema to add those declarations that are more specifically related to done events, except the `CookDone`. Therefore, `AvailableCook` is unaffected by this call. The item duration is extracted from the set of `directions` to determine when this call ends.

EventDone

$$Done$$

$$\mathrm{dom}\,(AvailableCook \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableCook \rhd \{currTime\})$$
$$currTime' = currTime + second\,(directions\ eid).item$$

`CookDone` (Figure 2) is the schema for making an unavailable cook `Available` again. Hence, it does not act on items, or ingredients (and their compositions). It obtains the `preparer?` from `directions` and at the end of the call, updates `AvailableCook` with this person included.

`Bake` (Figure 3) expects the `oven` to be in `HeatedItem` and the `what?` to be in `AvailableIngredient`s, which will all later be removed from this set and become present in the `UsedIngredient`. The `tool?` becomes `Dirty` at this event call but is still `Heated`. The `directions` is updated using all of this event information.

`RBake` is a stronger version of `Bake` that checks against bad conditions and reports success if it can execute.

$$RBake \mathrel{\widehat=} (Bake \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

`BakeDone` (Figure 4) is the counterpart of the `Bake` event. It does not modify `AvailableItem` or `DirtyItem`. At the end of this event, `HeatedItem` no longer contains the `oven`. Besides, it adds a new `AvailableIngredient` that is of type `BAKED_INGREDIENT` and updates this ingredient's composition to include those that were used in the initial `Bake` call. None of the other ingredients' compositions change.

[1]The `EventHandler` works as a temporary storage for done calls.

Event

$Kitchen$
$\Delta Timer$
$preparer? : Cook$
$tool? : KITCHEN\_ITEM$
$duration? : MEASUREMENT$
$dir : Direction$
$cookTime, toolTime : \mathbb{N}$
$ename : ENAME$

$preparer? \in \mathrm{dom}\,(AvailableCook \rhd \{currTime\})$
$first\ duration? \in \{minute, hour\} \land second\ duration? \neq 0$
$\mathrm{dom}\,(AvailableCook \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableCook \rhd \{currTime\}) \setminus \{preparer?\}$
$EventCount' = EventCount + 1$
$directions' = directions \cup \{((ename, EventCount) \mapsto dir), ((cookDone, EventCount) \mapsto dir)\}$
$first\ duration? = minute \Rightarrow toolTime = preparer?.initAction + second\ duration?$
$first\ duration? = hour \Rightarrow toolTime = preparer?.initAction + 60 * second\ duration?$
$tool? \in AUTO\_ITEM \Rightarrow cookTime = preparer?.initAction$
$first\ duration? = minute \land tool? \notin AUTO\_ITEM \Rightarrow cookTime = preparer?.initAction + second\ duration?$
$first\ duration? = hour \land tool? \notin AUTO\_ITEM \Rightarrow cookTime = preparer?.initAction + 60 * second\ duration?$
$EventHandler' = EventHandler \cup \{((ename, EventCount) \mapsto currTime + toolTime),$
$\quad ((cookDone, EventCount) \mapsto currTime + cookTime)\}$
$currEvents' = currEvents$
$currTime' = currTime + preparer?.initAction$
$\forall i : Ingredient \bullet \mathrm{ran}\,(\{currTime'\} \lhd i.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i.composedOf)$

Figure 1: `Event` schema is one of the major components in the system. It is reused in many event calls. Its purpose is to check whether the preconditions of an event are satisfied and if so, it adds them to the `EventHandler`.

In real life, one generally uses ingredients and kitchen tools interchangeably when the ingredient is *in* the tool. In order to solve this ambiguity, in this language, each and every preparation is referred to using only the ingredient since in fact the intended behavior is on the food rather than the item itself. Thus, all done events only return ingredients.

Apart from this fact, in this system, any `DirtyItem` is no longer usable. Therefore, `Clean` (Figure 5) event is used to return those `DirtyItem`s to the `AvailableItem` pool. The only self-cleansing tool is the `faucet`, which is assumed to be always clean. More specifically, `Clean` takes a `Dirty` but non-`Heated` `tool?`[2]. The `faucet` becomes un`Available` during this call. `DirtyItem`, `HeatedItem`, `AvailableIngredient`, and `UsedIngredient` are still the same. The appropriate `directions` are prepared to be included in the `Timer`.

`RClean`, similar to `RBake`, is a stronger version of `Clean` that reports success if the error conditions do not happen.

$$RClean \mathrel{\widehat=} (Clean \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

`CleanDone` (Figure 6) updates `AvailableItem`s with the `faucet` and the `cleaned!` and removes `cleaned!` from `DirtyItem`s. `HeatedItem` and `AvailableIngredient` remain unchanged as well as ingredients' composition fields.

`Cut` (Figure 7) schema uses `Available` and non-`Heated` `tool?` and `where?`, which stand for the item that is used to cut and another auxiliary item (like a cutting board) that helps in accomplishing this, respectively. These items cannot be the `counter` or the `faucet`. `what?` is a set of `AvailableIngredient`s. At the end time of this call, `AvailableIngredient`s no longer contain `what?` but instead `UsedIngredient`s do. As `tool?` and `where?` get added to `DirtyItem`, they are removed from `AvailableItem`. `HeatedItem` is unchanged throughout this time. `Knead` (Figure 11), `Mix` (Figure 13), and `Put` (Figure 15) events are similar to the `Cut` event and in fact, formally do not differ in their specifications; the only difference is user's intended system behavior, i.e. the visual effects. Therefore, these schemas are moved to Appendix A.

`Cut` is strengthened in the `RCut` schema. Due to the same reasons above, we moved `RKnead`, `RMix`, and `RPut` to Appendix A.

$$RCut \mathrel{\widehat=} (Cut \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

`CutDone` (Figure 8) schema is similar to `BakeDone` schema with two differences. First, it creates a `CUT_INGREDIENT` as an output as opposed to a `BAKED_INGREDIENT`. Second, and more importantly, it does not update `HeatedItem` because `Cut` does not operate on `HeatedItem`s. `KneadDone` (Figure 12), `MixDone` (Figure 14), and `PutDone` (Figure 16) are close in their specification to this schema and so are in the Appendix A.

The last schema in this section is `Preheat` (Figure 9) whose behavior is defined as follows. It takes an non-`HeatedItem` and `AvailableItem` that is the `oven` and makes it unavailable (but not `Dirty`) until it is finished with this `tool?`. It has no effect on `HeatedItem`, `AvailableIngredient`, and `UsedIngredient`. Also the `amount?` should be given in terms

[2]We do not want a burnt cook.

CookDone

$Done$
$preparer? : Cook$

$ename = cookDone \land (directions\ eid).c = preparer?$
$\text{dom}\,(AvailableCook \rhd \{currTime'\}) = \text{dom}\,(AvailableCook \rhd \{currTime\}) \cup \{preparer?\}$
$\text{dom}\,(AvailableItem \rhd \{currTime'\}) = \text{dom}\,(AvailableItem \rhd \{currTime\})$
$\text{dom}\,(DirtyItem \rhd \{currTime'\}) = \text{dom}\,(DirtyItem \rhd \{currTime\})$
$\text{dom}\,(HeatedItem \rhd \{currTime'\}) = \text{dom}\,(HeatedItem \rhd \{currTime\})$
$\text{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \text{dom}\,(AvailableIngredient \rhd \{currTime\})$
$\forall i : Ingredient \bullet \text{ran}\,(\{currTime'\} \lhd i.composedOf) = \text{ran}\,(\{currTime\} \lhd i.composedOf)$
$currTime' = currTime + preparer?.endAction$

Figure 2: `CookDone` schema is a done event thrown for each of `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` events. It brings the cook back to the table where s/he can work.

Bake

$Event$
$what? : \mathbb{P}\ Ingredient$

$ename = bakeDone \land what? \subseteq \text{dom}\,(AvailableIngredient \rhd \{currTime\})$
$tool? \in \text{dom}\,(HeatedItem \rhd \{currTime\}) \land first\ tool? = oven$
$\text{dom}\,(AvailableItem \rhd \{currTime'\}) = \text{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{tool?\}$
$\text{dom}\,(DirtyItem \rhd \{currTime'\}) = \text{dom}\,(DirtyItem \rhd \{currTime\}) \cup \{tool?\}$
$\text{dom}\,(HeatedItem \rhd \{currTime'\}) = \text{dom}\,(HeatedItem \rhd \{currTime\})$
$\text{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \text{dom}\,(AvailableIngredient \rhd \{currTime\}) \setminus what?$
$\text{dom}\,(UsedIngredient \rhd \{currTime'\}) = \text{dom}\,(UsedIngredient \rhd \{currTime\}) \cup what?$
$dir = \Theta\ Direction[c := preparer?, item := tool?, app := tool?, ingr := what?]$

Figure 3: `Bake` schema expects `AvailableIngredients` and an `Available` and `HeatedItem`. At the end of the call, the ingredients are `UsedIngredient` and the `tool?` is a `DirtyItem`.

of temperature.

Below is `RPreheat` schema that takes into account unexpected behavior in the `Preheat`.

$$RPreheat \mathrel{\widehat=} (Preheat \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

When `PreheatDone` (Figure 10) event is thrown, it returns the appliance back to `AvailableItem` but also adds it to `HeatedItem`. No changes occur in `DirtyItem` or `AvailableIngredient`.

4. ALLOY MODEL IMPLEMENTATION

In this section, we describe our steps to convert our Z specification into an Alloy model. Our goal in writing this model is to describe some aspects of our system (but not the entire system), to constrain it to exclude ill-formed examples, and automatically check properties about it. We paid special attention to finding and correcting errors in the Z specification that we are modeling and bugs in the Alloy model itself. Our Alloy model directly followed our Z specification with the following modifications:

1. `Measurement` or any constant is not modeled.
2. Each class of ingredients are represented with a static field called `IngredientType` that are used to represent `Baked`, `Cut`, `Kneaded`, `Mixed`, `Processed`, and `Raw` types.
3. `EventHandler` is not defined in the model.
4. A fact imposes an ordering on states such that it makes the state to directly jump to the next `Time` if the preconditions are met.
5. We also impose the following fact statements:
   (a) Every `CookDone` event must follow one of `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` events.
   (b) Every `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` event is respectively followed by one of `BakeDone`, `CleanDone`, `CutDone`, `KneadDone`, `MixDone`, `PreheatDone`, and `PutDone` events and also one `CookDone` event after it.
   (c) Every `BakeDone`, `CleanDone`, `CutDone`, `KneadDone`, `MixDone`, `PreheatDone`, and `PutDone` event respectively follows one of `Bake`, `Clean`, `Cut`, `Knead`, `Mix`, `Preheat`, and `Put` events before it.
6. `Cut`, `Knead`, `Mix`, and `Put` predicates are combined together in one predicate since they have exactly the same declarations.
7. `BakeDone`, `CutDone`, `KneadDone`, `MixDone`, and `PutDone` predicates are defined as one predicate since they change the state of the world in the same manner.
8. `Clean` can work on both dirty and clean items.
9. Error handling is unnecessary in the Alloy model.
10. Although our Z specification may work as different threads for each event (since it gives no clues about the

BakeDone

EventDone
$baked! : BAKED\_INGREDIENT$

$ename = bakeDone$
$\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\})$
$\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\})$
$\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \setminus \{(directions\ eid).item\}$
$\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \cup \{baked!\}$
$\forall i : Ingredient \bullet i \in (directions\ eid).ingr$
$\quad \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd baked!.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd baked!.composedOf) \cup \{\{i.iname\}\}$
$\quad \land\ i \notin (directions\ eid).ingr \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd i.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i.composedOf)$

Figure 4: `BakeDone` schema updates the `AvailableIngredients` to include a `BAKED_INGREDIENT` that is a composition of previously `UsedIngredients`.

Clean

Event
$where? : KITCHEN\_ITEM$

$first\ where? = faucet \land ename = cleanDone$
$tool? \in \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \land tool? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$where? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land where? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{where?\}$
$\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\})$
$\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\})$
$\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\})$
$dir = \Theta\, Direction[c := preparer?, item := tool?, app := where?, ingr := \emptyset]$

Figure 5: `Clean` schema makes it possible to reuse `DirtyItems` so, it has no effect on the ingredients.

implementation), Alloy model can process one event at a time.

11. Without loss of generalization, `Ingredient` signature has a `Sequence` (not a `set`) of `Ingredient`s as its composition rather than just their names.

With this implementation, we checked the following assertions that are essential to correct execution of our system:

1. No cooks are born or dead.
2. `Preheat` event call always precedes `Bake`.
3. Every cook becomes busy after event calls.
4. Item is the same tool after getting cleaned, i.e. no item disappears if all are cleaned.
5. Items become un`Available` after event calls.
6. `Preheat` is the only way to heat an item.
7. When `Done` events are thrown, items become `Available` again.
8. Ingredients become `UsedIngredient`s after event calls.
9. With `Done` events, new ingredients become `Available`.
10. `UsedIngredient`s never become `AvailableIngredient` again.
11. `HeatedItem`s should always be `Available` for use to protect against fire.
12. `AvailableIngredient`s are composed of only `UsedIngredient`s.
13. `Raw` ingredients are never a composition of other ingredients.
14. `AvailableIngredient`s that are not `Raw` are compositions of other ingredients.

These checks in our model we believe brought more confidence to our system that bad situations are less likely to happen.

5. DISCUSSION

In this section we will discuss some of our design decisions and some discoveries from our implementation.

Throughout this paper, we employed a direct Z definition to Alloy implementation order. However, although we in fact started with Z and later developed our Alloy model, the final Z and Alloy models were developed almost side-by-side. Our attempts showed that without our automatic analysis tools (both Z/Eves and Alloy Analyzer), proofs of completeness and soundness by hand would have required a substantial amount of effort and time. Also, their correctness would still be arguable. We particularly find type checking in Z/Eves and visualization and Evaluator components of Alloy very useful.

Because the level of automation in assisted theorem provers is still relatively poor compared to model checkers, our experiences with Z/Eves Theorem Prover showed that proofs in such systems are completed haphazardly. On the contrary, one of the great benefits of Alloy is its support of incremen-

CleanDone

$$
\begin{array}{l}
EventDone \\
cleaned! : KITCHEN\_ITEM \\
\hline
ename = cleanDone \land cleaned! = (directions\ eid)\ .item \\
\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \cup \{(directions\ eid)\ .app, cleaned!\} \\
\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \setminus \{cleaned!\} \\
\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \\
\forall i : Ingredient \bullet \mathrm{ran}\,(\{currTime'\} \lhd i\ .composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i\ .composedOf)
\end{array}
$$

Figure 6: `CleanDone` schema adds a `DirtyItem` to `AvailableItem`.

Cut

$$
\begin{array}{l}
Event \\
where? : KITCHEN\_ITEM \\
what? : \mathbb{P}\ Ingredient \\
\hline
ename = cutDone \land what? \subseteq \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \\
tool? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land tool? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
where? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land where? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
first\ tool? \notin \{counter, faucet\} \land first\ where? \notin \{counter, faucet\} \\
\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{tool?, where?\} \\
\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \cup \{tool?, where?\} \\
\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \setminus what? \\
\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\}) \cup what? \\
dir = \Theta\ Direction[c := preparer?, item := tool?, app := where?, ingr := what?]
\end{array}
$$

Figure 7: `Cut` schema makes use of `tool?` and `where?` items, and a set of `what?` ingredients. It updates the states of `AvailableItem`, `DirtyItem`, `AvailableIngredient`, `UsedIngredient` accordingly. `Mix`, `Knead`, and `Put` have similar schema declarations.

tal analysis. We initially explored design ideas starting from a tiny model (developed using the Z specification) and then scaled it up with making sure that Alloy is properly handling the extensions at every step.

Because we did not use Object-Z [4], class hierarchies were difficult to model in the Z specification as well as those types that require recursive nature[3]. Furthermore, tuples that use more than two types make it extremely cumbersome to extract their parts. Thus, we chose to define schemas as types since we want to have immediate access to their fields by name. Conversely, Alloy readily distinguishes between events (predicates) and states (signatures and facts), which are not transparent in the Z specification (all represented with schemas). It also easily captured the predicates that have overlapping definitions and required us to put them together into a single predicate.

Our Z specification can handle integers and constants. However, Alloy's capabilities are limited because a fully automatic analysis sacrifices completeness and can only find counterexamples that violate the constraints of the system within a limited scope[4]. Since integers are infinite, they cannot be properly modeled in Alloy. Because of this, our model can operate on discrete time points in which the event takes place. Similarly, as the number of constants increase, Alloy's analysis takes longer to finish. Therefore, we had to simplify our Z specification to still be able to illustrate the system behavior properly but comply with Alloy's limitations. Another big challenge is to match the events with their `Done` counterparts without the explicit use of an `EventHandler`.

[3]Ingredient is a type that links to used ingredients.

[4]The Small Scope Hypothesis [9] states that small scope checks are extremely valuable for finding errors.

6. CONCLUSIONS & FUTURE WORK

In this paper, we presented our attempts to formalize a novice programming language based on the cooking theme using formal language specification methods and automatic analysis tools. Although this mini-language is straightforward to interpret, it is important to make sure that bugs and errors do not exist since later the implementation will be based on it. Our final Alloy model is complicated and long ($\approx$ 500 lines) compared to the examples we have seen so far.

We also discussed some of the lessons learned from designing such a full specification and a model. The future work is to implement the language using this specification. We anticipate that it would not have been possible to directly build this surprisingly difficult language at the first attempt. Our insights that we gained through this work will later be valuable in the actual language development steps.

7. REFERENCES

[1] M. Addison. Best Toys 2008 – Hi-Tech Leads the Way. http://ezinearticles.com/?Best-Toys-2008—Hi-Tech-

CutDone

$$
\begin{array}{l}
\textit{EventDone} \\
\textit{cut}! : \textit{CUT\_INGREDIENT} \\
\hline
\textit{ename} = \textit{cutDone} \\
\mathrm{dom}\,(\textit{AvailableItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{AvailableItem} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{DirtyItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{DirtyItem} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{HeatedItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{HeatedItem} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{AvailableIngredient} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{AvailableIngredient} \rhd \{\textit{currTime}\}) \cup \{\textit{cut}!\} \\
\forall\, i : \textit{Ingredient} \bullet i \in (\textit{directions eid})\,.\textit{ingr} \\
\qquad \Rightarrow \mathrm{ran}\,(\{\textit{currTime}'\} \lhd \textit{cut}!\,.\textit{composedOf}) = \mathrm{ran}\,(\{\textit{currTime}\} \lhd \textit{cut}!\,.\textit{composedOf}) \cup \{\{i\,.\textit{iname}\}\} \\
\quad \land\, i \notin (\textit{directions eid})\,.\textit{ingr} \Rightarrow \mathrm{ran}\,(\{\textit{currTime}'\} \lhd i\,.\textit{composedOf}) = \mathrm{ran}\,(\{\textit{currTime}\} \lhd i\,.\textit{composedOf})
\end{array}
$$

Figure 8: `CutDone` schema creates a `CUT_INGREDIENT` and modifies it such that it contains the ingredients that were used to make it. Items remain the same.

Preheat

$$
\begin{array}{l}
\textit{Event} \\
\textit{amount}? : \textit{MEASUREMENT} \\
\hline
\textit{ename} = \textit{preheatDone} \land \textit{first tool}? = \textit{oven} \\
\textit{tool}? \in \mathrm{dom}\,(\textit{AvailableItem} \rhd \{\textit{currTime}\}) \land \textit{tool}? \notin \mathrm{dom}\,(\textit{HeatedItem} \rhd \{\textit{currTime}\}) \\
\textit{first amount}? \in \{\textit{fahrenheit}, \textit{celsius}\} \land \textit{second amount}? \neq 0 \\
\mathrm{dom}\,(\textit{AvailableItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{AvailableItem} \rhd \{\textit{currTime}\}) \setminus \{\textit{tool}?\} \\
\mathrm{dom}\,(\textit{DirtyItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{DirtyItem} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{HeatedItem} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{HeatedItem} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{AvailableIngredient} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{AvailableIngredient} \rhd \{\textit{currTime}\}) \\
\mathrm{dom}\,(\textit{UsedIngredient} \rhd \{\textit{currTime}'\}) = \mathrm{dom}\,(\textit{UsedIngredient} \rhd \{\textit{currTime}\}) \\
\textit{dir} = \Theta\,\textit{Direction}[c := \textit{preparer}?, \textit{item} := \textit{tool}?, \textit{app} := \textit{tool}?, \textit{ingr} := \emptyset]
\end{array}
$$

Figure 9: `Preheat` schema takes a non-`HeatedItem` and `AvailableItem` and removes it from the `AvailableItems` at its finish time. Ingredients are unaffected as well as `DirtyItems`.

Leads-the-Way&id=1672663, November 2008.

[2] Best Toys 2009 – Find the Top Toys and Hot Toys for Kids. http://besttoysguide.com/, March 2009.

[3] P. Brusilovsky. Turingal – the language for teaching the principles of programming. In *Proceedings of Third European Logo Conference*, Parma, Italy, 1991.

[4] D. A. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In *FORTE '89: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 281–296, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[5] A. Cockburn and A. Bryant. Leogo: An Equal Opportunity User Interface for Programming. *Journal of Visual Languages and Computing*, 8(5-6):601–619, 1997.

[6] M. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA, 1997.

[7] N. DS. Cooking mama: Cook off. http://www.cookingmamacookoff.com/, March 2007. A video game for the Wii.

[8] GameSpy. Cooking mama: Cook off review. http://wii.gamespy.com/wii/cooking-mama-cooking-with-international-friends/776739p1.html, March 2007.

[9] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.

[10] A. Karmel. *Mom and Me Cookbook*. Dorling Kindersley Children, August 29, 2005.

[11] M. Katzen and A. Henderson. *Pretend Soup and Other Real Recipes: A Cookbook for Preschoolers & Up*. Tricycle Press, April 1994.

[12] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A Sneak Preview. In *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, 2004.

[13] S. K. Nissenberg. *The Everything Kids' Cookbook: From mac'n cheese to double chocolate chip cookies – all you need to have some finger lickin' fun*. Adams Media Corporation, October 2002.

[14] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[15] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. John Wiley and Sons, New York, 1981.

[16] M. Resnick. Multilogo: A Study of Children and

PreheatDone

EventDone
preheated! : *KITCHEN_ITEM*

$ename = preheatDone \land preheated! = (directions\ eid).item$
$\text{dom}\,(AvailableItem \rhd \{currTime'\}) = \text{dom}\,(AvailableItem \rhd \{currTime\}) \cup \{preheated!\}$
$\text{dom}\,(DirtyItem \rhd \{currTime'\}) = \text{dom}\,(DirtyItem \rhd \{currTime\})$
$\text{dom}\,(HeatedItem \rhd \{currTime'\}) = \text{dom}\,(HeatedItem \rhd \{currTime\}) \cup \{preheated!\}$
$\text{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \text{dom}\,(AvailableIngredient \rhd \{currTime\})$
$\forall i : Ingredient \bullet \text{ran}\,(\{currTime'\} \lhd i.composedOf) = \text{ran}\,(\{currTime\} \lhd i.composedOf)$

Figure 10: `PreheatDone` schema prepares a `Heated` and `Available oven` so that a `Bake` event can later be called.

Concurrent Programming. *Interactive learning environments*, 1(3):153–70, 1990.

[17] M. Resnick. Starlogo: an environment for decentralized modeling and decentralized thinking. In *CHI '96: Conference companion on Human factors in computing systems*, pages 11–12, New York, NY, USA, 1996. ACM.

[18] M. Saaltink. The Z/Eves System. *ZUM '97: The Z Formal Specification Notation*, 1212:72–85, 1997.

[19] D. Sanders and B. Dorn. Jeroo: A Tool for Introducing Object-Oriented Programming. *SIGCSE Bull.*, 35(1):201–204, 2003.

[20] J. M. Spivey. *The Z Notation*. Prentice Hall International (UK) Ltd, second edition, 1992.

[21] S. Tarkan, V. Sazawal, A. Druin, E. Foss, E. Golub, L. Hatley, T. Khatri, S. Massey, G. Walsh, and G. Torres. Designing a Novice Programming Environment with Children. *University of Maryland Technical Report*, HCIL-2009-03, January 2009.

[22] I. Tomek. *The First Book of Josef: an introduction to computer programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[23] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

APPENDIX

A. EXTRA SCHEMAS

Because some of the schemas had the same declarations with minor changes, we put them into this section. `Knead` (Figure 11), `Mix` (Figure 13), and `Put` (Figure 15) schemas are similar to the previously described `Cut` (Figure 7). In addition, `KneadDone` (Figure 12), `MixDone` (Figure 14), and `PutDone` (Figure 16) are similar to `CutDone` (Figure 8) that was already explained in detail. Below, we also provide `RKnead`, `RMix`, and `RPut` schemas as the robust versions of `Knead`, `Mix`, and `Put`, respectively.

$$RKnead \mathrel{\widehat=} (Knead \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

$$RMix \mathrel{\widehat=} (Mix \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

$$RPut \mathrel{\widehat=} (Put \land Success) \lor UnknownCook \lor UnknownTool \lor UnknownIngredient \lor WrongDirection$$

Knead

$$
\begin{array}{l}
Event \\
where? : KITCHEN\_ITEM \\
what? : \mathbb{P}\, Ingredient \\
\hline
ename = kneadDone \land what? \subseteq \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \\
tool? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land tool? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
where? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land where? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
first\ tool? \notin \{counter, faucet\} \land first\ where? \notin \{counter, faucet\} \\
\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{tool?, where?\} \\
\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \cup \{tool?, where?\} \\
\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \setminus what? \\
\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\}) \cup what? \\
dir = \Theta\, Direction[c := preparer?, item := tool?, app := where?, ingr := what?]
\end{array}
$$

Figure 11: `Knead` **schema**

KneadDone

$$
\begin{array}{l}
EventDone \\
kneaded! : KNEADED\_INGREDIENT \\
\hline
ename = kneadDone \\
\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \\
\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \\
\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \cup \{kneaded!\} \\
\forall i : Ingredient \bullet i \in (directions\ eid).ingr \\
\qquad \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd kneaded!.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd kneaded!.composedOf) \cup \{\{i.iname\}\} \\
\quad \land\ i \notin (directions\ eid).ingr \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd i.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i.composedOf)
\end{array}
$$

Figure 12: `KneadDone` **schema**

Mix

$$
\begin{array}{l}
Event \\
where? : KITCHEN\_ITEM \\
what? : \mathbb{P}\, Ingredient \\
\hline
ename = mixDone \land what? \subseteq \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \\
tool? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land tool? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
where? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land where? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
first\ tool? \notin \{counter, faucet\} \land first\ where? \notin \{counter, faucet\} \\
\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{tool?, where?\} \\
\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \cup \{tool?, where?\} \\
\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\}) \\
\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \setminus what? \\
\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\}) \cup what? \\
dir = \Theta\, Direction[c := preparer?, item := tool?, app := where?, ingr := what?]
\end{array}
$$

Figure 13: `Mix` **schema**

MixDone

EventDone
$mixed! : MIXED\_INGREDIENT$

$ename = mixDone$
$\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\})$
$\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\})$
$\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \cup \{mixed!\}$
$\forall i : Ingredient \bullet i \in (directions\ eid).ingr$
$\quad \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd mixed!.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd mixed!.composedOf) \cup \{\{i.iname\}\}$
$\quad \land\ i \notin (directions\ eid).composedOf \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd i.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i.composedOf)$

Figure 14: `MixDone` **schema**

Put

Event
$where? : KITCHEN\_ITEM$
$what? : \mathbb{P}\ Ingredient$

$ename = putDone \land what? \subseteq \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\})$
$tool? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land tool? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$where? \in \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \land where? \notin \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$first\ tool? \notin \{counter, faucet\} \land first\ where? \notin \{counter, faucet\}$
$\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\}) \setminus \{tool?, where?\}$
$\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\}) \cup \{tool?, where?\}$
$\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \setminus what?$
$\mathrm{dom}\,(UsedIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(UsedIngredient \rhd \{currTime\}) \cup what?$
$dir = \Theta\ Direction[c := preparer?, item := tool?, app := where?, ingr := what?]$

Figure 15: `Put` **schema**

PutDone

EventDone
$processed! : PROCESSED\_INGREDIENT$

$ename = putDone$
$\mathrm{dom}\,(AvailableItem \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableItem \rhd \{currTime\})$
$\mathrm{dom}\,(DirtyItem \rhd \{currTime'\}) = \mathrm{dom}\,(DirtyItem \rhd \{currTime\})$
$\mathrm{dom}\,(HeatedItem \rhd \{currTime'\}) = \mathrm{dom}\,(HeatedItem \rhd \{currTime\})$
$\mathrm{dom}\,(AvailableIngredient \rhd \{currTime'\}) = \mathrm{dom}\,(AvailableIngredient \rhd \{currTime\}) \cup \{processed!\}$
$\forall i : Ingredient \bullet i \in (directions\ eid).ingr$
$\quad \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd processed!.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd processed!.composedOf) \cup \{\{i.iname\}\}$
$\quad \land\ i \notin (directions\ eid).ingr \Rightarrow \mathrm{ran}\,(\{currTime'\} \lhd i.composedOf) = \mathrm{ran}\,(\{currTime\} \lhd i.composedOf)$

Figure 16: `PutDone` **schema**