Controller Synthesis: From Modelling to Enactment

Víctor Braberman*, Nicolas D'Ippolito†, Nir Piterman‡, Daniel Sykes†, Sebastian Uchitel*†
* Universidad de Buenos Aires, Argentina, {vbraber, suchitel}@dc.uba.ar
† Imperial College London, UK, {srdipi, daniel.sykes}@imperial.ac.uk
‡ University of Leicester, UK, nir.piterman@le.ac.uk

***Abstract*—Controller synthesis provides an automated means to produce architecture-level behaviour models that are enacted by a composition of lower-level software components, ensuring correct behaviour. Such controllers ensure that goals are satisfied for any model-consistent environment behaviour. This paper presents a tool for developing environment models, synthesising controllers efficiently, and enacting those controllers using a composition of existing third-party components.**

Video: www.youtube.com/watch?v=RnetgVihpV4

***Index Terms*—Controller Synthesis, LTS, Reactive Planning.**

I. Introduction

Controller synthesis is receiving increased attention in the software engineering community as a means for producing architecture-level behaviour models that can be enacted by a software system to ensure correct-by-construction behaviour.

For instance, synthesis of glue code and component adaptors has been studied in order to achieve safe composition at the architecture level [1], and in particular in service-oriented architectures [2]. More recently, there has been an increasing interest in self-adaptive systems [9] which must be capable of designing adaptation strategies at run-time. Consequently, such systems rely heavily on automated synthesis of behaviour models that will guarantee the satisfaction of requirements under the constraints enforced by the environment and the capabilities offered by the self-adaptive system.

Controller synthesis requires a behaviour model (e.g. a Labelled Transition System – LTS) of the environment and a goal (e.g. a formula in linear temporal logic – LTL) to be achieved by the software system. Controller synthesis delivers a behaviour model which if enacted by the software is guaranteed to achieve its goals for any behaviour the environment may exhibit (and that is consistent with the provided behaviour model).

Controller synthesis is in general computationally expensive (for LTL, 2EXPTIME complete [14]). Nevertheless, restricting the form of the goal allows for polynomial time solutions. For example, goal specifications consisting uniquely of safety requirements can be solved in polynomial time, and so can particular styles of liveness properties such as GR(1) [13].

In this paper we describe a tool that extends MTSA [6] (which provides sophisticated features for developing and validating environment behaviour models [11], [15], [8], [7], [17]) to *i)* support controller synthesis for an expressive goal language [13], fallible domains [4] and partial environment models [5], and to *ii)* integrate with an enactment framework running on the Backbone middleware [12].

II. Motivating Example

Consider a scenario in which we have a robot arm, a drill, a painting tool, an in tray, an out tray, a recycle bin and some additional sensors. The arm can be moved freely and there is no fixed connection between any tool or tray. We aim to use the arm aided by sensors to move objects from the in tray, through a combination of tools according to a high-level production process, to the out tray.

The control mechanisms of the robot arm are provided through a general purpose API with support for moving the arm to a specific coordinate, and for identifying and grabbing objects. In addition, due to traction issues, the arm may fail when trying to grab an object, a situation that can be sensed through an operation that reads whether the gripper is fully closed or not.

Our approach allows for *i)* development of higher-level operations which are programmed on top of the API (e.g. a parameterised pick up operation that moves the arm to the specified tool/tray and attempts up to three times to identify an object of a specified colour and grab it); *ii)* modelling the environment in terms of these operations (e.g. picking up from the drill can only succeed if an object was previously placed there, the paint tool paints objects red); *iii)* specification of a high-level production process (e.g. produce alternating coloured objects $(red.yellow)^\star$, only painted objects that have been drilled); *iv)* specification of environment assumptions (e.g. yellow objects will be supplied indefinitely, the probability of successfully grabbing an object is greater than 0); *v)* synthesis of a controller in the form of a behaviour model that encodes the arm's strategy for achieving the production process (e.g. what to do if it needs to output a red object but is receiving only yellow objects); and finally *vi)* running the arm based on the synthesised controller by calling the high-level operations provided and monitoring the events the arm senses.

III. Environment Modelling

The semantic basis for reasoning about the behaviour of the environment and controllers are Labelled Transition Systems (LTSs) [10], which are widely used for modelling and analysing the behaviour of concurrent and distributed

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Formal Demonstrations

```
PAINT = (putdownat_success['paint][Colours]
        ->COLORING),
COLORING = (ready['paint]['red]
    ->pickupfrom_success['paint]['red]->PAINT)
+{ready['paint]['yellow],
pickupfrom_success['paint]['yellow]}.
TOOL(T='any)=(putdownat_success[T][c:Colours]
    ->ready[T][c]->pickupfrom_success[T][c]->TOOL).
||TOOLS = (forall[t:Tools] TOOL(t)).
ARM = (pickupfrom[l:GetLocations][c:Colours]
    ->GET_RESULT[l][c]),
GET_RESULT[l:GetLocations][c:Colours]=
        (pickupfrom_success[l][c]->PICKED_UP[c]
        | pickupfrom_fail[l][c]->ARM),
PICKED_UP[c:Colours]=(putdownat[l:PutLocations][c]
        ->putdownat_success[l][c]-> ARM).
SUPPLY = (supply[c:Colours]
        ->pickupfrom_success['in][c]->SUPPLY).
||SUPPLIER = SUPPLY.
FORCE_PICKUP=(supply['yellow]->COUNT[0]
            | A\{supply['yellow]}->FORCE_PICKUP),
COUNT[id:Count] = (A\{pickupfrom['in]['yellow]}
                ->COUNT[id+1]
          |pickupfrom['in]['yellow]->FORCE_PICKUP),
COUNT[MAX+1] = (pickupfrom['in]['yellow]
                ->FORCE_PICKUP).
||ENV = (SUPPLIER || PAINT || TOOLS
          || ARM || FORCE_PICKUP).
```

Fig. 1. FSP Example

systems. An LTS is a state transition system where transitions are labelled with actions.

We build on MTSA [6] for environment modelling where an extension of the Finite State Processes (FSP) language is used. FSP is a textual language with strong emphasis on the compositional construction of complex models used to describe LTSs [11]. FSP includes several traditional operators for describing behaviour models, such as action prefix (`->`), choice (`|`), sequential composition (`;`), and parallel composition (`||`). Extensions support modelling partial behaviour models [7] and use of scenario and declarative specifications (e.g. [15])

In Figure 1 we show a snippet of the tool with the FSP code describing the behaviour of the robot arm, tools and objects supply (see Section II).

The FSP process `ARM` models the fact that picking objects up from a location can fail. As expected, the arm must successfully pick up an object to be able to put it somewhere else. `PAINT` models the behaviour of the painting tool which paints red any object it is given. `TOOLS` is a parametric model that captures the behaviour of tools that receive objects, and only after their processing is done they allow for objects to be picked up. `SUPPLY` models that objects are only supplied when the in tray is empty. Finally, the environment model (`ENV`) is the result of the parallel composition of the LTSs modelling the robot arm, the tools, and the supplier.

The tool provides support for validating and verifying environment models using graphical animation [17] and model checking [8]. However, such features are out of the scope of this paper.

IV. Controller Synthesis

In addition to an environment model, the tool requires the specification of goals the controller is expected to achieve in

```
OUT_PROTOCOL = (putdownat_success['out]['red]
        -> putdownat_success['out]['yellow]
        -> OUT_PROTOCOL).
ltl_property TOOL_ORDER =
        [] (F_PAINT -> F_HAVE_DRILLED)
||PICK_UP_IF_OBJECT_PRESENT =
                    (PICK_IN_PRE
                    || PICK_PAINT_RED_PRE
                    || PICK_PAINT_YELLOW_PRE
                    || PICK_TOOLS_PRE).
fluent FFAILED_PICKUPS =
            <FAILURE_SET, A\{FAILURE_SET}>
assert YELLOW_IN = F_SUPPLY_YELLOW
assert RED_OUT = F_COLOUR_PUT['out]['red]
controllerSpec RED_YELLOW = {
    safety = {OUT_PROTOCOL,
            TOOL_ORDER,
            PICK_UP_IF_OBJECT_PRESENT}
    failure = {FAILED_PICKUPS}
    assumption = {YELLOW_IN}
    liveness = {RED_OUT}
    controllable = {CA}
}
controller ||C = (ENV)~{RED_YELLOW}.
checkCompatibility ||COMP = (ENV)~{RED_YELLOW}.
```

Fig. 2. Controller Goals - FSP Example

Fluent Linear Temporal Logic (FLTL) [8]. Linear temporal logics (LTL) are widely used to describe behaviour requirements [8]. The motivation for choosing an LTL of fluents is that it provides a uniform framework for specifying state-based temporal properties in event-based models [8]. FLTL is a linear-time temporal logic for reasoning about fluents. A *fluent* fl is defined by a set of initiating actions I_{fl}, a set of terminating actions T_{fl}, and an initial value $Initially_{fl}$. In Figure 2, the property `TOOL_ORDER` specifies that objects must be drilled before they are painted.

The tool supports SGR(1)-like goals, that is, goals of the form $\Box I \wedge (\bigwedge_{i=1}^{n} \Box\Diamond A_i \rightarrow \bigwedge_{j=1}^{m} \Box\Diamond G_j)$ where $\Box I$ is the safety part of the controller goals, $\Box\Diamond A_i$ represents a liveness assumption on the environment behaviour, $\Box\Diamond G_j$ models a liveness goal for the controller and A_i and G_j are non-temporal fluent expressions [8], while I is a safety property expressed as a Fluent Linear Temporal Logic formula [8].

In addition, the tool supports controller synthesis in the presence of fallible domains [4]. In other words, it is capable of dealing with environments in which controlled actions may fail as long as failures are probabilistic in nature or, more formally, it is possible to assume *strong independent fairness* [4]. Solving a control problem under this assumption can be reduced to SGR(1) [4] and hence remains polynomial. The events assumed to be strongly independent fair are specified simply as a set of fluents which indicate when a failure has occurred. In Figure 2, the expression `FAILED_PICKUPS` describes failures for the example in Section II.

The full specification for the controller to be synthesised is described with the `controllerSpec` keyword requiring `safety` ($\Box I$) and `liveness` goals (G_j), the liveness `assumption` for the environment (A_i), the declaration of (`failures`), and the set of actions that are `controllable`. See Figure 2 where `OUT_PROTOCOL` requires alternating coloured objects, `TOOL_ORDER` requires painting only drilled objects, and `PICK_UP_IF_OBJECT_PRESENT` requires attempting to pick up only if a object is at the location.

In Figure 2 the `controller` operator returns a behaviour model that satisfies the controller specification for a given environment model if such a controller exists. The `checkCompatibility` operator checks if the assumptions on the environment are realisable by the environment, if this is the case the model returned by `controller` is guaranteed to not be *anomalous* [3]. In other words, the controller will not attempt to prevent the environment from satisfying the assumptions in order to avoid having to satisfy the controller goals. For example, if the `FORCE_PICKUP` process is removed from `RED_YELLOW` in Figure 1 then `ENV` is not compatible. This is because the controller could never pick up an object from the in tray preventing the environment to provide yellow objects. Note that an alternative way of achieving a compatible control problem is to allow the environment to enqueue objects in the in tray.

Although we cannot show the controller due to space restrictions (it has over 5000 states, compared to the over 10000 states of the environment model) we describe some of the behaviour its exhibits: : i) If it is the turn to output a red objects but the environment provides a yellow one, the controller has to drill it, then paint it red and finally put it down on the out tray. ii) When it is the turn to output yellow ones but the environment provides a red one, then the controller can assume that at some point a yellow product will be supplied and, hence, discard the current red product and wait for the next yellow to appear. iii) When it is the turn for red products to be produced, if the controller is processing (i.e. drilling, then painting) a yellow to get a red but a red is supplied, the controller may choose to discard the yellow being processed and just output the red waiting in the in tray.

V. Enactment

In general, we wish to create abstract operations that hide the complexity presented by the API of the low-level software components of the system, thus producing an alphabet of high-level actions for use in the environment model and controller. As the components are normally provided by third parties such as the robot arm manufacturer, each high-level action may map to an *ad hoc* combination of low-level method calls. The designer must provide Java implementations of each high-level action (or event) that will be used in the environment model. These implementations consist of combinations of method calls with appropriate parameters on the existing third-party components. The components run on the Backbone middleware [12], and configurations of them are automatically assembled using our previous work on adaptive self-assembly [16].

Figure 3 depicts the framework's architecture. The two main domain-independent components are the *interpreter* and the *configuration manager*. The interpreter executes a controller by keeping track of the current state, executing controlled actions (using the domain-specific implementations) and responding to environment events. More specifically, when the current state is controlled, the interpreter selects one of the enabled actions at random. When the current state is uncontrolled, or a mixed controlled/uncontrolled state, the interpreter waits to receive an environment event.

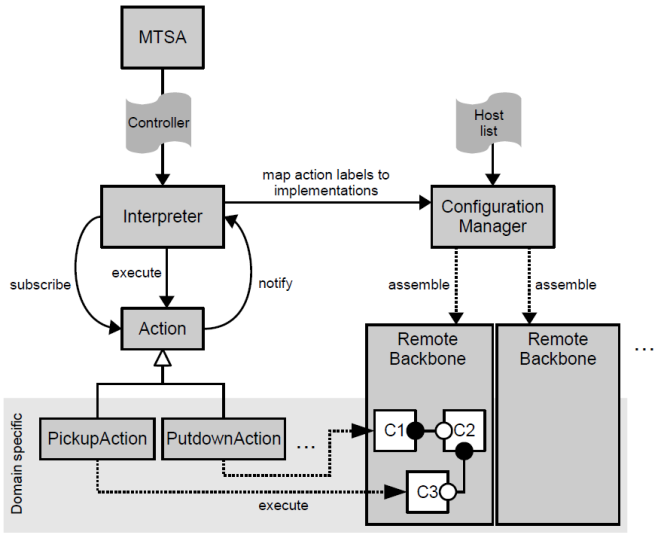

Fig. 3. Architecture of the framework. Dashed arrows represent potentially remote calls.

Each controlled action is implemented by a subclass of `Action` (following the Command pattern). Additionally, each such implementation can generate environment events. Figure 4 shows part of an implementation of a `pickupfrom` action, in which an existing component method is called (`pickupFrom`) with appropriate parameters. The action label includes a location name and a colour name. The location name must be converted to the API-defined type `KatanaLocation`, which represents a point in the physical space of the robot arm. This is done by looking up the location name in a table of known locations. Conveniently, the colour parameter can be passed straight through. Note that we have omitted code that checks the action label is "well formed", with the right number of parameters within allowed ranges. The success of the action is determined by calling another component method (`gripperClosed`). The implementation then generates either a success or a failure event. Notice that the implementation is largely contained inside a separate thread. This is because the interpreter should be allowed to transition to the next state of the controller before waiting for the time-consuming physical operation `pickupFrom` to complete.

The set of action implementations available to the interpreter is determined by the configuration manager. Before beginning execution, the configuration searches for implementations of each action (and environment event) and instantiates them reflectively. The configuration manager is aware of a number of potentially-remote Backbone hosts and the third-party components available that are available on each host. The configuration manager then queries each action implementation to determine which components it requires during execution, and checks which of the Backbone hosts is capable of assembling a configuration containing those components. If all requirements are found, then the configuration manager instructs each

```
1  public class PickupFromAction extends Action {
2    public void execute(ActionLabel action) throws ExecutionException {
3      final String location = action.getParameters().get(0);
4      final String colour = action.getParameters().get(1);
5      final RemoteBallGrabber grabber =
6        (RemoteBallGrabber) configMan.getInterfaceInstance(component);
7      Thread runner = new Thread("PickupFrom") {
8          public void run() {
9            KatanaLocation loc = lookupLocation(location);
10           grabber.pickupFrom(loc, colour);
11           boolean succeeded = !grabber.gripperClosed();
12           if (succeeded)
13             eventHappened(new Event(new ActionLabel("pickupfrom_success."
14                     +location+"."+colour)));
15           else
16             eventHappened(new Event(new ActionLabel("pickupfrom_fail."
17                     +location+"."+colour)));
18         }
19       };
20     runner.start();
21   }
22 }
```

Fig. 4. Fragment of code implementing a `pickupfrom` action.

Backbone to assemble its components, following the approach set out in [16]. Finally, each action implementation calls the configuration manager to get a (remote) reference to the components that are called, as in line 3 of Figure 4.

In effect, the resulting system is a correct-by-construction orchestration of third-party components which satisfies the safety and liveness goals given in the MTSA model specification, and which handles certain types of failure.

VI. Conclusions

We have presented an overview of the main features of an extension of the MTSA tool that supports controller synthesis and enactment, implementing the body of work that we have developed in the last years related to adaptive systems, controller synthesis and behaviour modelling. A video of the tool at work is available at `http://youtu.be/RnetgVihpV4`. The tool itself and examples are available at `http://sourceforge.net/projects/mtsa/`.

References

[1] M. Autili, P. Inverardi, M. Tivoli, and D. Garlan. Synthesis of "correct" adaptors for protocol enhancement in component-based systems. In *Specification and Verification of Component-Based Systems*, page 79. ACM, 2004.

[2] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[3] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran. Softw. Eng. Methodol.*, 22, 2013.

[4] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behaviour models for fallible domains. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 211–220. ACM, 2011.

[5] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. The modal transition system control problem. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 155–170. Springer, 2012.

[6] N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 475–476, Washington, DC, USA, 2008. IEEE Computer Society.

[7] D. Fischbein, N. D'Ippolito, G. Brunet, M. Chechik, and S. Uchitel. Weak alphabet merging of partial behavior models. *ACM Trans. Softw. Eng. Methodol.*, 21(2):9, 2012.

[8] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[9] A.-C. Huang, D. Garlan, and B. Schmerl. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proceedings of the First International Conference on Autonomic Computing*, pages 276–277, Washington, DC, USA, 2004. IEEE Computer Society.

[10] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, July 1976.

[11] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Wiley New York, 2006.

[12] A. McVeigh, J. Kramer, and J. Magee. Using Resemblance to Support Component Reuse and Evolution. In *Proc. of SIGSOFT/FSE Workshop on Specification and Verification of Component-based Systems*, New York, NY, USA, 2006. ACM Press.

[13] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Lecture notes in computer science*, 3855:364–380, 2006.

[14] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[15] G. Sibay, V. Braberman, J. Kramer, and S. Uchitel. Synthesising modal transition systems from triggered scenarios. *IEEE Transactions Software Engineering*, PP(99):1, 2012.

[16] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From Goals to Components: A Combined Approach to Self-Management. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'08, 2008.

[17] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Goal and scenario validation: a fluent combination. *Requir. Eng.*, 11(2):123–137, 2006.