SYNTHESIZING EXECUTABLE PROGRAMS FROM REQUIREMENTS

by

Cory Plock

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2008

Benjamin Goldberg

Dedication

For my beloved wife Lizette, who gave me the strength to continue when the end wasn't anywhere in sight. Also for my parents, whose love, support, and inspiration have guided me through the past many years. Finally, for my brother Michael, whose love, encouragement, and sense of humor have propelled me through all the tough times.

Acknowledgments

I would like to thank my advisor Ben Goldberg and co-advisor Hillel Kugler for their guidance, support, and encouragement over the past many years. I would like to also extend my gratitude to Amir Pnueli for taking the time to provide research guidance, and to Anina Karmen for going far above and beyond the call of duty. Thanks to Bonnie MacKellar, who introduced me to the world of research as an undergraduate and always challenged me to excel. Finally, thank you to Mr. Charles McFarlane, who saw talent in me that I couldn't see myself, and without whom I couldn't have reached this point.

Abstract

Automatic generation of correct software from requirements has long been a “holy grail” for system and software development. According to this vision, instead of implementing a system and then working hard to apply testing and verification methods to prove system correctness, a system is rather built correctly by construction. This problem, referred to as synthesis, is undecidable in the general case. However, by restricting the domain to decidable subsets, it is possible to bring this vision one step closer to reality.

The focus of our study is reactive systems, or non-terminating programs that continuously receive input from an external environment and produce output responses. Reactive systems are often safety critical and include applications such as anti-lock braking systems, auto-pilots, and pacemakers. One of the challenges of reactive system design is ensuring that the software meets the requirements under the assumption of unpredictable environment input. The behavior of many of these systems can be expressed as regular languages over infinite strings, a domain in which synthesis has yielded successful results.

We present a method for synthesizing executable reactive systems from formal requirements. The object-oriented requirements language of Live Sequence Charts (LSCs) is considered. We begin by establishing a mapping between various subsets of the language and finite-state formal models. We also consider LSCs which can express time constraints over a dense-time domain. From one of these models, we show how to formulate a winning strategy that is guaranteed to satisfy the requirements, provided one exists. The strategy is realized in the form of a controller which guides the system in choosing only non-violating behaviors. We describe an implementation of this work as an extension of an existing tool called the Play-Engine.

Contents

List of Figures

List of Tables

Introduction

The general consensus among those in the software engineering community is that object-oriented programming—particularly its role in enabling software reuse—is a favorable programming paradigm. Many practitioners are attracted to the idea that one can write software, test it, and then deliver it to another developer without the recipient having to concern herself with implementation details. The recipient can instead simply refer to “the interface,” which usually amounts to a textual description of what the object is supposed to do.

Creating objects for other developers to use is a process which lends itself to intra-object design practices. That is, designs which focus primarily on the data members of the object and how the data changes over time with respect to method calls. In fact, considerable effort goes into testing individual objects. However, comparatively little attention seems to be paid to inter-object issues in practice—that is, the question of whether objects properly interact with each other.

This gives rise to large libraries of objects whose inter-object behavior is unclear, despite a clear understanding of each individual object’s behavior. The inevitable consequence is software which does not behave as expected. This could possibly be explained, in part, by the fact that popular modeling languages such as UML are weak on expressing object interactions. Arguably, intra-object constructs such as Statecharts tend to be taken more seriously among developers than the inter-object counterparts, e.g., sequence diagrams.

The problem of arriving at correct software extends beyond the inter-object vs. intra-object design issue. Indeed, even in the presence of perfect designs, any implementation process that incorporates human intervention on a large scale is virtually guaranteed to introduce errors into the final implementation. Errors can range from behavior which was not intended by the requirements author(s) to straight-out system crashes. We therefore find it reasonable to explore the possibility of removing at least some of the human component of software implementation.

Software Lifecycle

The software development process begins with the *requirements* phase, referring to the gathering of information which describes how a final program is intended to behave. The output of the *specification* phase delivers a set of software blueprints, describing how the requirements are to be carried out. The *implementation* phase refers to the process of writing instructions (e.g., in a programming language), part of which often involves the compilation to machine level code. The *testing* phase amounts to formal or informal verification of the implemented software with respect to the requirements.

A correct executable program is the desired outcome of this process. A correct program, with respect to the requirements, carries out all intended behaviors and no unintended ones. In order to be classified as incorrect, a program need only carry out just one unintended behavior or fail to carry out one intended behavior.

Incorrect behavior has the potential to be catastrophic in the case of safety critical software systems. Past incidents confirm that our fears are justified. For example, energy management software—later found to be incorrect [Jes04]—failed to alert officials of a problem that contributed to the largest rolling power outage in the history of the northern United States in August, 2003. In September 1999, a relatively simple unit conversion error caused the untraceable disappearance of NASA's Mars Climate Orbiter, marking a total economic loss of over 327 million dollars [NAS99].

Researchers and engineers have struggled for years to develop software design and implementation methodologies which lead to correct executable programs. Despite numerous useful research results and programming methodologies that have emerged over the years, a general solution for consistently achieving correctness remains elusive for all but the most trivial programming tasks.

Requirements can take on a variety of forms ranging between prose to a formal description. In practice, most fall somewhere in between. One popular requirements methodology is the Object Management Group's standardized Unified Modeling Language (UML)[BRJ99], and more recently, UML 2.0.

One typical problem with requirements is *incompleteness*, characterized by their inability to classify all possible behaviors as good or bad. For example, requirements are often biased toward *sunny day scenarios*, whereby "normal" behavior is described in full detail, but exceptional behavior—most often caused by unexpected input—is not. Failure to account for unexpected inputs at the requirements level leads to behavior whose correctness is unknown. In some cases (e.g., software crashes) these behaviors are obviously bad. Other cases are more subtle in the sense that some may view the behaviors as correct, while others may not. In the absence of a precise blueprint, it is impossible to determine.

Even when the requirements are complete, the final implementation can still nevertheless be incorrect. Experience shows that this is attributable to human error in many cases, and can be traced to any phase of the software life-cycle which involves human intervention. Specific causes can include poor coding technique, miscommunication, incorrect operating assumptions, language barriers, or a multitude of other factors. Also, requirements in written form may not necessarily agree with the intentions of the person who wrote them—the designer may have overlooked certain subtleties in the description, for example.

Lacking any better methods at their immediate disposal, design teams often just try their best at writing correct code with the hope that creating and executing test cases will capture most errors. While better solutions may exist, whether it be in practice or in theory, most software engineers lack knowledge of formal verification techniques.

Of particular interest to us are *reactive systems*, or non-terminating programs which perpetually respond to environment inputs [HP85]. These systems are found everywhere and include toasters, microwaves and alarm clocks, for example. Many safety critical applications are reactive too, including anti-lock braking systems, pacemakers, and aircraft autopilots—all of which, if not correct, can lead to catastrophic consequences.

In most reactive systems, assumptions about external environment behavior must necessarily be limited. By this, we mean that one cannot assume that the environment will cooperate in delivering inputs precisely as the software designer expected. These systems are environment-driven insofar as they are expected to respond to the environment, not vice versa. The burden is therefore on the system to respond to each environment input correctly, in the sense that the totality of these behaviors—those of both the system *and* environment—satisfies the requirements.

In this work, we consider a subset of the object-oriented requirements language Live Sequence Charts (LSCs) [DH01]. LSCs express inter-object behaviors using *scenarios*, where emphasis is placed on the interaction between objects in the system. This is in contrast to the more usual intra-object approach which focuses on the internal behaviors of each object. One of our reasons for choosing LSCs is that behaviors can be described visually, in a way that many people—even non-developers—can understand. The visual aspect of LSCs does not come at the cost of precision or clarity, as as LSCs describe reactive behaviors precisely.

An ambitious vision is presented in [Har00], whereby fully executable code could someday be generated seamlessly from requirements. According to this vision, the object-oriented scenario-based requirements language of LSCs would also serve as the final implementation. The research community has responded rather positively to this vision by proposing a number of approaches to reaching this goal. See, for example, [HM01, KW01, BH02, BS03b, Gil03, PGZ05].

Contributions

We proceed according to the above vision by presenting a method for synthesizing executable reactive systems directly from requirements. Our work is broken down into two major parts.

We first present a method for the automatic translation of requirements to specifications. That is, a translation from the visual domain of LSCs to deterministic Büchi automata. We then extend the result to include single LSC models which can express rational time constraints over a dense time domain, and present a method for translating from this domain to deterministic timed Büchi automata. We conclude with one of the main contributions of this work, a translation from multiple-LSC requirements over a dense time domain to deterministic timed Büchi automata.

The second part discusses two solutions for performing controller synthesis on LSCs, starting from a specification. The first is an algorithm which abstracts away certain specification details to focus on the general framework for solving the problem intuitively for LSCs. Secondly, we describe an approach for modifying an existing specification to be suitable for input to a controller synthesis algorithm. We also describe a modification to the original result which permits us to express LSC behaviors more naturally than we could otherwise.

A more detailed discussion of our contributions appears in Chapter 7, after the reader has become familiarized with the material in this work.

Chapter 1

Requirements Languages

The first stage in the software life-cycle is the creation of *requirements* that express a design team's behavioral expectations of a future application. Our goal is to automatically translate requirements to executable code, so it is essential that we only consider requirements which are precisely defined. A *requirements language* is used for authoring requirements, similar to the way a programming language is used to write programs.

In this work, we restrict our attention to specifying *reactive systems* [PR89b, PR89a], or systems which continuously react to external stimuli. This is in contrast to *computational* programs which, given an input, produce some output and then terminate. Reactive systems are characterized by an infinite sequence of *environment inputs* and system *reactions*.

1.1 Synchronous Languages

We adopt the *synchrony hypothesis*, a notion first described in [BG92]. The hypothesis is best explained by the paper itself:

> "Synchrony amounts to saying that the underlying execution machine takes no time to execute the operations involved in instruction sequencing, process handling, inter-process communication, and basic data handling."

The main purpose of adopting the synchrony hypothesis is to abstract away these details in order to focus on behaviors which are more relevant to the requirements set forth by the user. Requirements languages operating under this hypothesis are commonly referred to as *synchronous languages*.

The first synchronous languages were developed independently in the 1980s and early 1990s, most notably Esterel [BMR83], Lustre [HCRP91], Signal [GBGM91], and Argos [Mar90]. Prior to these languages, deterministic automata, Petri-Nets and concurrent programming languages were primarily used as requirements. None of the prior solutions were very succinct. Automata, for example, could easily grow beyond a size that is conducive to human understanding. Moreover, systems comprised of multiple components generally required asynchronous semantics, giving rise to other problems such as components "interrupting" each other during reactions.

The synchronous languages above are mostly text-based and exhibit the flavor of a programming language. They all have precise semantics, are relatively succinct, and abstract away irrelevant behavioral details.

1.2 Visual Modeling

One drawback to the synchronous languages mentioned above is that they can sometimes be *too* succinct. A complete reactive system specification may appear to have a small amount of input, but at the cost of readability. For example, minor changes to the program input can dramatically change the output behaviors. Even worse, input changes may alter the output behaviors more subtly, in a way that eludes the designer. Therefore, larger more complex input programs can easily become unmanageable in practice.

An alternative approach is to describe behaviors *graphically* instead. Statecharts [Har87] is one such methodology which has enjoyed great commercial success. It exhibits the simplicity of finite state automata but addresses the size issue by injecting the notions of state-hierarchy, concurrency, and communication into the previously flat automaton model. A number of software tools have been developed to support Statecharts, most notably, Statemate [HLN$^+$90] and Rhapsody [HG96, HK04].

1.2.1 Message Sequence Charts

Another visual language is Message Sequence Charts. Originally standardized in 1992 [Z1293], Message Sequence Charts (MSCs) is a formal graphical requirements language originally used by the telecommunications industry for modeling the behaviors of distributed telecommunications software. MSCs were the original inspiration for *sequence diagrams* which were later adopted into the popular Unified Modeling Language (UML)[BRJ99].

MSCs offer an *inter-object* or *scenario-based* approach to behavioral modeling.

Rather than specifying the behavior of each object individually, MSCs model the interaction *between* objects. The individual object behavior is then inferred from the interaction.

Each MSC scenario is depicted by a Basic Message Sequence Chart (bMSC), which visually depicts the participating *instances* in a scenario and the interactions between them. Under the synchrony hypothesis, atomic behaviors called *events* occur in zero time. The most basic form of interaction is the sending and receiving of asynchronous *messages* between objects over a reliable FIFO channel. Message sending and receiving are examples of events.

Each instance has a vertical *instance line*, representing a temporal top-to-bottom ordering of behaviors. Each instance line has an ordered set of *locations*, and there exists a surjective mapping from locations to events. The permissible ordering of events over one instance is a strict ordering of the locations on the instance line. The permissible ordering over multiple instances is a partial order over the locations. A bMSC *run* is a maximal linearization of the partial order.

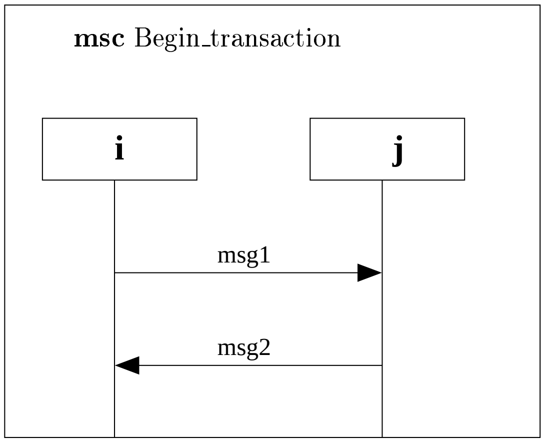


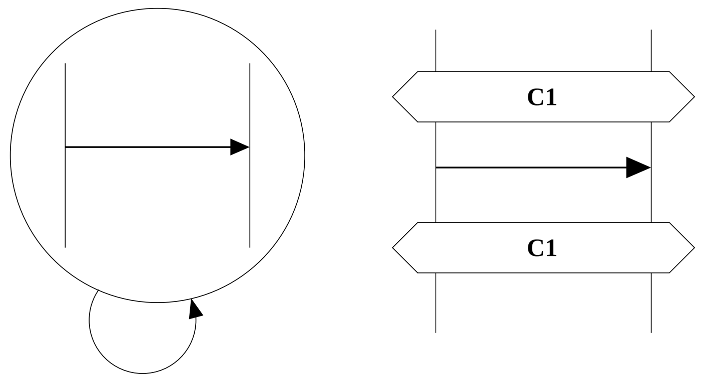
Figure 1.1: Basic Message Sequence Chart

Consider the MSC of Fig. 1.1, entitled Begin_transaction. This scenario models the interaction between two instances, *i* and *j*. The horizontal arrows depict asynchronous *messages* sent from one instance to the other. According to this scenario, instance *i* sends message $msg1$, sometime after which instance *j* receives it. Another message, $msg2$ is communicated reciprocally.

MSC syntax can be converted into a textual BNF grammar, allowing one to reason formally about the structure. An extension of the original MSC standard [Z1293] provides a translation from the textual representation to a process algebra, which is intended to assign semantics to the language. The precision and scope of these semantics had been an issue of debate in the literature [LL93, Ren95]. Assuming a complete and unambiguous process algebra, the BNF grammar would imply the regularity of the MSC language in the sense that it can be converted into an automaton for the purpose of formal reasoning, such as verification.

1.2.2 Multiple MSCs

Most interesting systems are too complex to describe in a single bMSC. This gives rise to the question of how multiple MSCs can be combined to form complete requirements. This question is first addressed in the revised *MSC96* standard [Z1296], which offers two general solutions: The first proposes a flow graph called a High Level MSC (hMSC) that connects bMSCs using arrows. An hMSC may also contain other hMSCs, producing a hierarchical requirements structure. The second is *composition*, whereby continuation points located inside bMSCs depict the entry or exit points for other bMSCs. Fig. 1.2 illustrates the two methods.



(a) (b)

Figure 1.2: MSC Requirements using hMSCs and Sequential Composition.

High Level MSCs depict a flow of execution between vertices, each vertex denoting either a bMSC or another hMSC. An initial and terminal vertex is assumed. The vertices may connected—using branching choice, concatenation, and iteration—thereby inducing a language that expresses the entirety of the requirements. Fig. 1.2(a) shows a simple example of hMSCs using iteration.

The composition approach utilizes a condition, depicted by a hexagon, to denote the continuation of a scenario. For instance, when the bottom condition in Fig. 1.2(b) is reached, it synchronizes with the similarly labeled condition on the top, admitting an infinite execution.

1.2.3 Shortfalls

The MSC example in Fig. 1.2(a) describes a non-regular language. Every send event must be paired with an eventual receive event and the FIFO queue can contain arbitrarily many send events, giving rise to the issue of unboundedness. This could occur, for example, in a situation where messages have been sent but communication delays prevent them from being received right away.

Although a graphical means of specifying behavior could be useful for increasing readability, as we have just seen, MSCs unfortunately suffer from the same problems synchronous languages already addressed. In a sense, this is a step backward. Although problems such as the above have been addressed [HMNT99, HMKT00], the user is still forced to deal with these extraneous details.

Most software executes certain behaviors conditionally. It is therefore necessary on the requirements level to have a mechanism for specifying that certain behaviors are allowed or disallowed based on the truth values of conditions. The MSC standard suggests overloaded semantics for conditions. Conditions serve the dual purpose of restricting behaviors (*guarding conditions*) and making assertions over the state space (*setting conditions*). In the latter case, the condition may possibly double as a composition operator.

The process algebra suggested by the ITU standard unfortunately does not provide semantics for conditions in either case. Although variations have been suggested, none seem to have gained mainstream acceptance, making the absence of condition semantics a fairly significant drawback.

It has been argued in the literature that safety properties alone are insufficient for describing most reactive behavior. For example, Damm & Harel point out that MSCs lack *liveness*. That is, the ability to specify whether behaviors *can* happen or *must* happen. It is also unclear whether MSC conditions *can* or *must* evaluate to true. That is, whether false conditional evaluations are a violation of the requirements, or whether they merely affect the subsequent behaviors.

1.3 Live Sequence Charts

Live Sequence Charts (LSCs) [DH01] is an extension of MSCs which remedies the above shortcomings by adding liveness to the requirements, among other things. By assigning a hot or cold *temperature* to MSC messages, conditions, and locations, LSCs can describe whether events are mandatory or provisional. Additionally, they can describe whether or not scenarios themselves are mandatory, and if so, the conditions under which they must occur.

LSCs come in two varieties: *universal* and *existential*. Universal LSCs restrict the set of permissible behaviors by imposing requirements that must hold over all system runs. They can be used to describe scenarios that must happen and even *anti-scenarios*—those which must not happen. Existential LSCs, on the other hand, impose requirements which must hold over at least one system run. They can be viewed as the dual of universal LSCs: whereas universal LSCs *restrict* behaviors from occurring, existential LSCs *require* behaviors to occur.

Instances on an LSC are each identified by a name, which appear across top of the LSC within solid rectangles. Extending downward from these are *instance lines* which depict the flow of control for each instance in the scenario. Each instance executes the behaviors depicted on its instance line in a top-down manner. Universal charts consist of a *prechart*, annotated by a dashed hexagon, and a *main chart*, annotated by a solid rectangle. The relationship between the prechart and main chart is one of temporal causality: if the behaviors of the prechart occur then the behaviors of the main chart are required to eventually occur.

LSC messages can depict a variety of real-life behaviors. For example, a message could represent a packet of transmitted data, the invocation of a function call, or a real-world event (e.g., sunrise). The type of event is usually understood by context. For our purposes, it is usually not necessary to make the distinction.

1.3.1 Example

Consider Fig. 1.3, an example of LSC requirements consisting of precisely one scenario, described by a universal LSC. The scenario describes the requirements of a simple cellular phone power-on sequence. Three instances, `User`, `Power Button` and `Display`, participate in the scenario. Specifically, `User` represents the external environment and the other two instances belong to the cellular phone. In general, there can exist other scenarios belonging to the same requirements, which could describe additional behaviors over and above that of Fig. 1.3. For now, we focus our attention on single LSC requirements.

The prechart of Fig. 1.3 depicts a *synchronous message* called $Press$ which is sent from `User` to `Power Button`. Synchronous messages, denoted by a solid arrow head, are characterized by the instantaneous occurrence of sending and receiving. In the example, `User` sends $Press$ at the same instant as `Power Button` receives it. In real life, the message describes the act of physical touch—the sender and receiver of the message refer to the initiator and recipient of the action, respectively. An *asynchronous message*, not shown in this example, is depicted in the same way as synchronous mes-

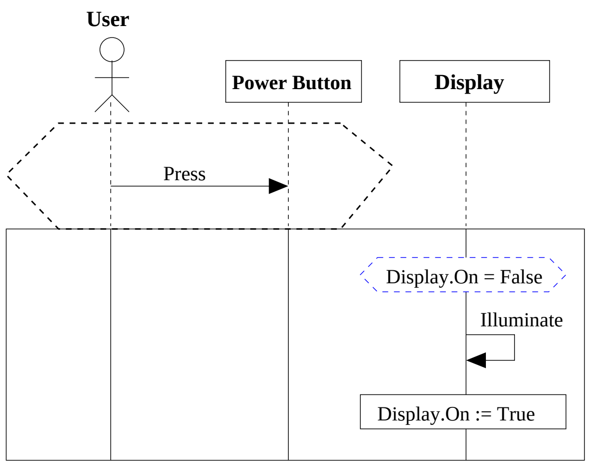


Figure 1.3: Cellular Phone Scenario: Turn On

sages, except for an open arrowhead. Asynchronous messages are suitable for describing events in which a delay occurs between the sending and receiving.

LSCs may contain *LSC variables* and *object properties*, both of which are similar to variables in many imperative programming languages. They are both typed and refer to memory locations wherein discrete values, corresponding to the type, can be stored. LSC variables are an attribute of the LSC, whereas object properties are an attribute of an object. LSC variables are initially undefined each time the scenario begins executing, and are visible and mutable only within one particular LSC. In contrast, object properties are persistent between scenario executions and are visible to and mutable by any LSC in the requirements.

LSC variables and object properties can participate in LSC assignments and conditions. An assignment is depicted by a small solid rectangle and an embedded annotation of the form $\varphi := x$ where φ is an LSC variable or object property and x is a constant. A condition is depicted as a small solid hexagon annotated by a predicate over the LSC variables and object properties. Based on the valuation of the variables or object properties, conditions evaluate to *true* or *false* in the obvious way.

When an LSC condition evaluates to true, the scenario proceeds beyond the condition. When it evaluates to false, the consequence depends on the *temperature* of the condition. Temperature is depicted by coloring the condition box red or blue, corresponding respectively to a *hot* or *cold* temperature. Hot conditions are *required* to evaluate to true—a false evaluation constitutes a violation of the requirements. A false evaluation of a cold condition, on the other hand, causes the scenario to terminate without violation[1]. The condition shown in Fig. 1.3 is cold, meaning that the scenario will proceed to the bottom of the main chart if the display is not on. Otherwise, the chart will terminate without violation. Intuitively, this means that the scenario is applicable only to the case where the display is not on.

Putting it all together, the scenario of Fig. 1.3 states that whenever the user presses the power button, it is required that either the `Display` must already be on, or otherwise must illuminate itself and then switch to an On state. This scenario handles only the case of the phone being turned on, even though the power button might serve as an on/off toggle.

We expand the above requirements by adding the LSC of Fig. 1.4, which describes the scenario of turning off the phone. The LSC consists of a prechart which is identical to that of Fig. 1.3 and a main chart which is similar. The conditional expression at the top of Fig. 1.3 is the negation of the corresponding prechart in Fig. 1.3. Specifically, if the `Display` instance of Fig. 1.4 is in an On state, then it executes the $PowerDown$ message and satisfies the main chart by setting the On variable to false. The LSCs

[1]We assume in this work that LSCs do not contain subcharts. Generally speaking, however, if a cold condition within a subchart evaluates to false then only the subchart terminates.

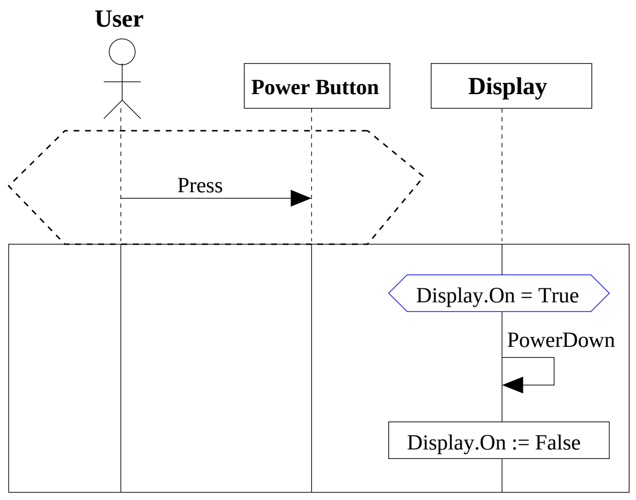


Figure 1.4: Cellular Phone Scenario: Turn Off

of Figs. 1.3 and 1.4 work in conjunction with each other to describe the entire on/off switching behavior. When `User` sends the *Press* message, both precharts are simultaneously satisfied and the `Display` instance is then required to satisfy both main charts. If the phone's display is already off, then the "Phone Off" scenario in Fig. 1.4 immediately ends due to the condition. However, the "Phone On" scenario of Fig. 1.3 continues on to illuminate the display and then change the state of the display's *On* property to true.

1.3.2 Justification for Use

The requirements phase of a software development life-cycle is arguably the most important, since it has been shown through practice that careful design reduces errors and overall implementation time. The requirements phase is intended to capture the behavioral description of the software. There is a vast range of requirements languages ranging from informal descriptions to rigorous and complete formal modeling tools.

There exist contrasting viewpoints concerning the intended purpose of requirements, and the extent to which they should be used. Some practitioners view requirements as only an abstract description or set of general guidelines, rather than a concrete set of rules. According to this opinion, real-life requirements are too abstract and therefore insufficient for addressing the inherent complexities of a software system. Others adopt a stricter approach, citing that requirements should strive to express the most complete behavioral description possible, even if the state-of-the-art does not currently permit the level of rigor or formality necessary for a totally precise description. According to this point of view, an up-front reduction in the level of ambiguity results in a more guided development approach, thereby reducing the need for time-consuming discussions during the implementation phase.

This work is primarily directed to those who adopt the latter approach of utilizing formal and precise requirements languages to capture behavioral detail, or those willing to explore the possibilities of using this approach. We do not claim superiority of this approach over the former approach, or any other. However, we do maintain that the use of formal requirements languages have proven to be useful for a variety of reactive software projects.

In this work, we study the use of LSCs as a requirements language. Precise LSC definitions permit us to reason formally about LSC behavior for the purposes of synthesizing executable code. From a user standpoint, LSCs are related in appearance and function to UML sequence diagrams, with which many practitioners in the field are already familiar. We believe that a visual depiction of scenario-based behaviors is intuitive to not only professional developers, but also those with little or no technical background in software development. Existing tools such as the Play-Engine [HM03] also serve as an aid in capturing LSC requirements by generating LSCs through the user manipulation of a graphical interface.

Although introduced only within the past few years, LSCs have already shown to be promising for developing or modeling many types of systems. For example, they have been used to model the weather synchronization logic for NASA's Center TRACON Automation System [BHK03], to create virtual wrappers for a PCI bus [BG02], to design a radio-based train system [BDK$^+$02], and even to model living biological systems [KHK$^+$03]. We find these initial results encouraging for developing the embedded software systems in which we're interested.

Chapter 2

LSC Requirements

We now provide definitions for the subset of LSC requirements considered in this work. The subset is comprised of messages, assignments, conditions, variables, and properties. These definitions are similar to the more extensive treatments appearing in [HK00a, HM03]. We begin with a discussion of untimed LSCs and then expand the discussion in Chapter 3 to include the notion of time. We focus initially on requirements consisting of only one LSC and then later treat the more common and expressive case of multiple LSC requirements.

2.1 Object Systems

An *object system* is a set of objects which represent hardware, software, users, or other physical or conceptual entities that interact with one another. Each object acts on behalf of the *system* or *environment*. System objects are those belonging to and controlled by the application we seek to design, whereas environment objects and their behaviors are outside the system's control.

2.1.1 Definitions

Formally, an object system is a double $Sys = \langle \mathcal{D}, \mathcal{O} \rangle$ where $\mathcal{D}$ is a set of *application types* and $\mathcal{O}$ is a set of *objects*. Each application type $D \in \mathcal{D}$ is a finite set of values. We make a simplifying assumption that each set is strictly-ordered so that the binary comparison operators (e.g. $\leq, <, \geq, >$) can be applied to all types. This allows us to develop a single theory, rather than addressing the formal details of a complete type system. For types where ordering doesn't make sense, the comparison operators aren't used.

An object is a triple $\langle \mathtt{name}, \mathcal{P}, own \rangle$ where $\mathtt{name}$ is the name of the object, $\mathcal{P}$ is a set of properties (data members) and $own \in \{\mathit{sys}, \mathit{env}\}$ classifies the object's ownership as either system or environment, respectively. Properties are defined by $\langle \mathtt{name}, D, \omega, \Theta \rangle$, where $\mathtt{name}$ is the name of the property, $D \in \mathcal{D}$ is its application type, $\omega \in D$ is the *property value*, and Θ is the initial value.

We adopt the convention of referring to elements of a tuple using an object-oriented "dot" notation. For example, to distinguish between the set of properties belonging to objects O_i and O_j, we write $O_i.\mathcal{P}$ and $O_j.\mathcal{P}$, respectively. We also extend this notation to members of sets; for example, if we first establish the existence of a particular property $p \in O_i.\mathcal{P}$, we may later write $O_i.p$ to make the ownership explicit.

The set of all properties is given by $\mathit{Props} = \{p \mid p \in O.\mathcal{P} \ \wedge \ O \in \mathcal{O}\}$. The state space of the object system, denoted $\Sigma[\mathcal{O}]$, refers to the set of all valuations of properties in $Props$. The *initial valuation* of $\Sigma[\mathcal{O}]$, denoted $init(\Sigma[\mathcal{O}])$, is the valuation that maps all properties $p \in Props$ to their initial value, $p.\Theta$.

2.1.2 Relationship to LSCs

Intuitively, an object system can be viewed as the full set of components making up an executable program. Each component can communicate with other components by means of *messages,* which we refer to more abstractly as *behaviors*. Sending and receiving messages may cause the state of the objects (i.e., the property valuations) to change over time.

On a high level, LSCs model object system behaviors. They precisely describe acceptable and unacceptable behaviors. Given an object system and LSC requirements, we may check to see if any executable trace of that system agrees with the requirements. Alternatively, we may use the object system and requirements to automatically generate a "book of rules" by which the object system is guaranteed to agree with the requirements. The former method has already been implemented in a tool called the Play-Engine [HM03]. The latter method—also known as *synthesis*—is precisely the subject of this work, and we later describe an extension to the Play-Engine to support it.

2.2 Live Sequence Charts

Formally, a *Universal Live Sequence Chart* L over object system Sys is an 8-tuple defined by $L = \langle I, M, V, A, C, Pch, evnts, temp \rangle$ consisting of:

- I — a set of *instances*, each defined as a pair $\langle \texttt{locs}, O\rangle$ where $\texttt{locs}$ is a (finite) totally ordered set of locations and $O \in \mathcal{O}$ is the object corresponding to the instance. We denote the set of locations of an instance i by $\texttt{locs}(i)$. The location numbers across all instances are mutually distinct. We set $\texttt{locs}(L) = \bigcup_{i \in I} \texttt{locs}(i)$.

- M — a set of *messages* $\langle \texttt{name}, i_s, i_r, \ell_s, \ell_r, \texttt{mode}\rangle$ where $\texttt{name}$ is the name of the message, $i_s, i_r \in I$ are the sending and receiving instances, $\ell_s \in \texttt{locs}(i_s), \ell_r \in \texttt{locs}(i_r)$ are the locations at which the message is sent and received[1], and $\texttt{mode} \in \{a, s\}$ denotes whether the message is *asynchronous* or *synchronous*.

- V — a finite set of typed *LSC variables*, each defined by $\langle \texttt{name}, D, \omega\rangle$ where $\texttt{name}$ is the variable name, $D \in \mathcal{D}$ is the type, and $\omega \in D \cup \{\perp\}$ is the value. We use $\perp$ to denote an undefined variable. We require all types $\mathcal{D}$ to be finite and discrete. By $\Sigma[V]$ we denote the state space of V—that is, the set of valuations mapping each $v \in V$ to a value $v.\omega$. The *initial valuation* of V, denoted $\mathit{init}(\Sigma[V])$, is the valuation where all variables are undefined.

- A — a set of *assignments* $\langle \texttt{locs}, \texttt{expr}\rangle$, where $\texttt{locs}$ is the set of locations to which the assignment is anchored (i.e., visually attached to the location), and $\texttt{expr}$ is an expression of the form "$\varphi := x$" where $\varphi \in (V \cup \mathit{Props})$ and $x \in \varphi.D$. We denote the set of locations corresponding to a particular assignment $a \in A$ by $\texttt{locs}(a)$. If a is an assignment, we may refer to φ as $a.\varphi$ and x as $a.x$.

- C — a set of *conditions* $\langle \texttt{locs}, \texttt{expr}\rangle$, where $\texttt{locs}$ is the set of locations to which the condition is anchored. Conditions compare either variables or properties to a

[1]An instance may send a message to itself, in which case $i_s = i_r$ and $\ell_s = \ell_r$.

constant using an operator $\# \in \{=, <>, <, >, \leq, \geq, =, \neq\}$. In particular, expression `expr` is of the form "$\varphi \# x$" where $x \in v.D$.

- $Pch \subseteq \texttt{locs}(L)$ — a *prechart*.
- $evnts\colon \texttt{locs}(L) \rightarrow E$ — a many-to-one mapping between locations and *events*. The set of events, E consists of the union of visible events E_v, hidden events E_h, and unrestricted events E_u. Events are discussed in greater detail in section 2.5. Let $evnts^{-1} : E \mapsto 2^{\texttt{locs}(L)}$ map an event to the set of locations in the LSC where the it occurs.
- $temp\colon \texttt{locs}(L) \cup C \rightarrow \{H, C, W\}$ — maps a location to a hot, cold, or warm *temperature*. Also maps a condition to a hot or cold temperature. We turn to the topic of temperature in the next section.

2.3 Liveness

Each LSC instance depicts a visual top-to-bottom ordering of events, as represented formally by the instance's ordered set of locations. When an object system executes, each instance begins in its topmost location. During the course of execution, the LSC keeps track of the current location and moves to the next consecutive location whenever the event corresponding to that location occurs.

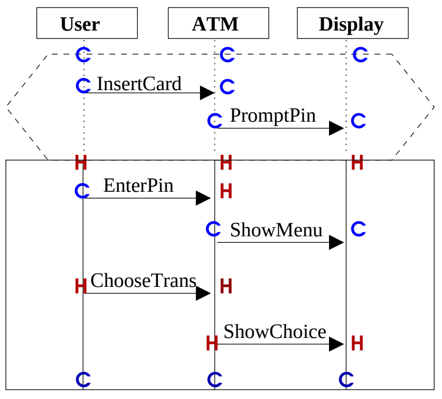
A central concept of LSCs is the notion of *liveness*, which is expressed by assigning a temperature to every location. The temperature of a location tells us whether the next consecutive event *can* or *must* occur. In the former case, the object system is given the choice of whether or not to execute the next event. In contrast, failure to ever execute it in the latter case is considered a violation of the requirements.

2.3.1 Hot and Cold Temperature

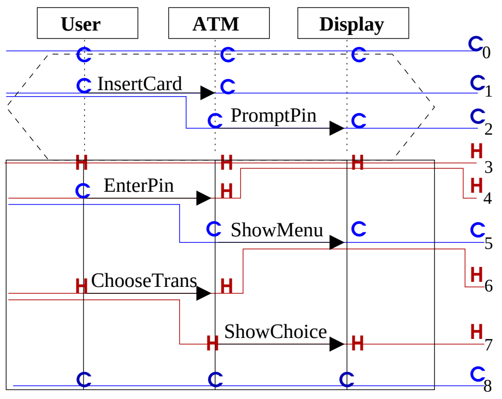
A cold location, depicted by a blue location coloring, represents that an instance may either remain forever at the location or progress. Hot locations, depicted by a red location coloring, impose a mandatory progress requirement. To avoid dependence on color displays, location temperatures in the following example and throughout the remainder of this work are depicted by the letters H, C, denoting hot and cold, respectively. Later we will discuss the *warm* temperature, which is depicted on the LSC by the letter W.

Later, in section 2.8.1, we formally discuss the concept of a *cut*, which is a snapshot of an LSC's configuration during execution. Roughly speaking, it is the product comprised of one location from every instance. Hot and cold temperatures extend to cuts in the following way. A *cold cut* consists of all cold locations, whereas a *hot cut* consists of at least one hot location. Similar to location temperatures, an LSC may remain forever in a cold cut without violating the requirements, whereas hot cuts require eventual progress.

Consider the scenario of Fig. 2.1(a) which illustrates the interaction between a user (shown as `User`), automatic teller machine (`ATM`), and a user display (`Display`). Fig. 2.1(b) shows the cuts overlaid on Fig. 2.1(a). For the purposes of this example, we number the cuts $0, \ldots, 8$ and show the temperature of each using letters H and C on the far right.



(a) ATM SCENARIO



(b) ATM SCENARIO WITH CUTS

Figure 2.1: LSCs with Fairness

The LSC is in precisely one cut at a time, starting at cut 1 and progressing sequentially. Cuts 0, 3, and 8 are special cuts which are not associated with any messages, the first and last of which are the *initial* and *final* cut, respectively. In order to reach these cuts, special *hidden events* must occur. Hidden events are discussed in further detail in section 2.8.6.

Fig. 2.1 requires that whenever a user inserts a card (cut 1) and the ATM prompts the user for a PIN on the display (cut 2), the prechart is satisfied (cut 3) and the main chart is consequently required to eventually be satisfied. In particular, when `User` sends $EnterPin$, the main chart progresses to cut 4. Since cut 4 is hot, the ATM is required to send the $ShowMenu$ to `Display`, causing the main chart to progress to cut 5. Since cut 5 is cold, no progress is required from this point. However, if `User` sends the $ChooseTrans$ message, the LSC progresses to cut 6—a hot cut—from which it is required that `ATM` eventually send the $ShowChoice$ message. When the message is sent, the LSC progresses to cut 7. The LSC can finally move to cut 8, thereby satisfying the LSC.

2.3.2 More on Cold Temperatures

The LSC literature describes two possible interpretations of cold temperatures. According to the first, instances take the lazy approach of never progressing from a cold cut. This interpretation is safe in the sense that no harm can be done (with respect to violating the requirements) by remaining forever in a cold cut. We refer to this as a *minimal* policy, as defined in [HM03].

The second interpretation states that instances should always progress from a cold cut if they can, despite the absence of a requirement to do so. This option is not as safe because sometime after leaving the cold cut, the possibility of a future violation arises. This interpretation follows a *maximal* policy.

Precharts contain only cold locations by which follow a maximal policy. There would be no need for a main chart otherwise, since it would never be reached under a minimal policy. Main charts, on the other hand, can follow a minimal or maximal policy. The Play Engine, in particular, follows a maximal policy for both precharts and main charts.

2.3.3 Warm Temperatures

Rather than assign two or more possible semantics to cold locations, we instead assign the minimal policy exclusively to cold temperatures and create a new *warm* temperature to express the maximal policy.

Aside from obtaining greater clarity, another advantage of this approach is that we permit the user to freely use cold, warm or hot temperatures within the main chart of universal LSCs. Whenever a sequence of one or more warm locations precedes a sequence of one or more hot locations, this is equivalent to stating that if all events leading up to the last warm location occur, then all the events associated with the subsequent hot locations must occur. For this reason, we require that all prechart locations are warm. A similar semantics is also suggested for Modal Sequence Diagrams in [HM07], whereby alternation between *cold fragments* (blocks of warm locations) and *hot fragments* (blocks of hot locations) is permitted.

We say that an event is *enabled* if it is legal for that event to occur next based on the sequence of events that has already occurred. We have already mentioned LSC cuts, which record the present position of each instance along an instance line, and hence the set of events which have already occurred. Therefore, we typically say that events are enabled or not enabled with respect to some cut. The notion of enablement is discussed more formally in section 2.8.2.

Cuts map to temperatures in the following way: if any location in the cut is hot, then the cut is hot. If no locations in the cut are hot, but at least one location is warm, then the cut is warm. Otherwise, the cut is cold. With these notions of cut temperature and enablement, we arrive at the following semantics for expressing LSC liveness properties:

- If an enabled event occurs from a warm cut, the LSC must progress from that cut.
- If a non-enabled event occurs from a warm cut, the LSC must remain forever in the warm cut.

We hereafter assume that all prechart locations are warm. Main chart locations may be either cold, warm, or hot, except for the locations appearing at the very bottom of the chart, which must be cold so that the chart closes. These assumptions do not affect the later results of this work, but are adopted only to remain consistent with the generally accepted LSC semantics.

If an LSC is in a warm cut and the next event causes a violation of the partial order, the LSC remains in the current cut forever. If the cut were cold, the decision to remain there forever would have been made upon reaching the cut. As described above, warm cuts permit the decision (of whether or not progress) to be delayed until the next event occurs. In either case, when the decision to remain forever in the cut has been determined, the LSC may close without violation since all obligations have been fulfilled. From a practical standpoint, it is wasteful to consume memory resources to perform bookkeeping for an LSC whose state will never change in the future.

We illustrate the use of warm locations using the example of Fig. 2.2. This example is identical to Fig. 2.1 except that certain cold locations have now been replaced by warm locations. In particular, we make all prechart locations warm and, additionally, change some of the main chart temperatures from cold to warm.

When message sequence $InsertCard$, $PromptPin$ occurs, the LSC progresses to cut 3 thereby satisfying the prechart. Since cut 3 is warm, the LSC is required to progress if $EnterPin$ occurs, but the main chart closes without violation if some other message occurs. Supposing that $EnterPin$ occurs, the LSC advances to cut 4—a hot cut. It is therefore required that $ShowMenu$ eventually occur. If it does, the LSC progresses to cut 5. Because cut 5 is warm, the LSC must progress if $ChooseTrans$ occurs or the LSC otherwise closes without violation. Supposing $ChooseTrans$ occurs, the LSC progresses to cut 6, from which there is a new requirement that $ShowChoice$ must eventually occur.

If it never does, the requirements are violated. Otherwise, the LSC advances to cut 8—a cold cut—which causes the LSC to close.

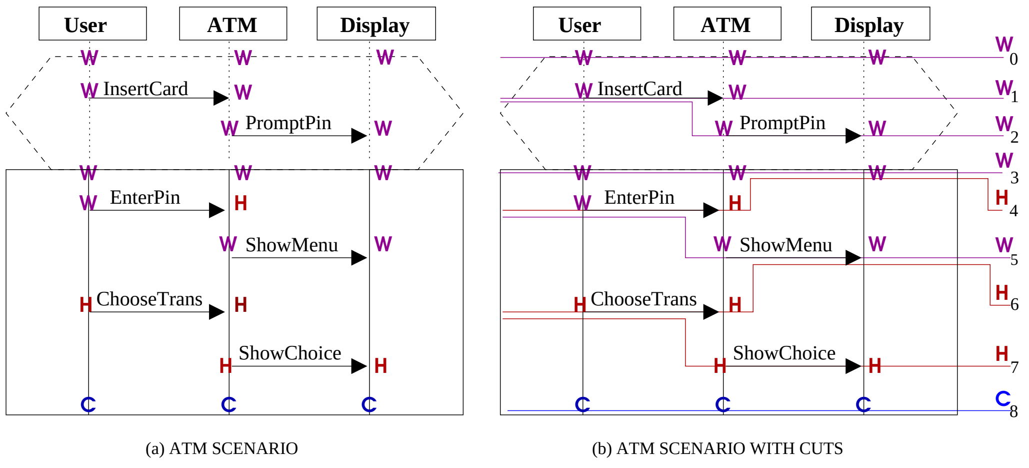


Figure 2.2: LSC with Warm Temperatures

Fig. 2.2 illustrates classic reactive behavior, where `User` is viewed as the external environment and `ATM`, `Display` belong to the system whose behavior we are specifying. We see a series of obligations created during the course of the run. The requirements of this example essentially state that if the system and environment cooperate in producing the message sequence $InsertCard$,$PromptPin$, then the following holds: if the environment sends $EnterPin$ then the system must eventually send $ShowMenu$. Thereafter, if the environment sends $ChooseTrans$ then the system must eventually send the message $ShowChoice$.

The above example describes a set of “if-then” behavioral requirements. Without warm locations, the above behavior could not be described using only one LSC. Roughly speaking, each “if” would require a prechart of its own, and therefore a distinct LSC.

2.3.4 Conditions

Conditions are depicted on LSCs as small hexagons with a dashed border. Inside the hexagon is a *conditional expression* over the LSC variables and object properties. When a specific valuation in $\Sigma[V] \cup \Sigma[\mathcal{O}]$ is applied to the expression, a Boolean output is produced, indicating whether the expression is true or false. Conditions are treated more formally in section 2.8.3, but we provide a brief overview of the semantics here.

Conditions are *anchored* to locations on one or more instance lines. Instance lines that are attached to a particular condition are said to *participate* in that condition. When all participating instances are exactly one location above the condition, the condition can be *evaluated*. If the evaluation is true, all instances move onto the condition simultaneously. The outcome of a false conditional evaluation depends on the condition's *temperature*, which is determined by the $temp$ function.

All conditions, similarly to locations, take on a hot or cold temperature[2]. Hot conditions are shown on the LSC by a red hexagonal border, whereas cold conditions are blue. We now explain the significance of condition temperature.

Each LSC scenario tells the story of a behavioral interaction among instances. The story begins with all instances at the topmost locations and ends when they arrive at the bottom. Sometimes its necessary to end the story prematurely, for example when the system is in a state from which it makes no sense to proceed. *Cold conditions* are used to abort scenarios in this way. When a cold condition evaluates to false, the LSC closes normally.

[2]There are no warm conditions.

In other cases, a requirements designer may wish to assert that it is unacceptable for the system to arrive at or remain in a particular state. A system that reaches such a state is said to be in violation of the requirements. *Hot conditions* are used to perform the assertion. If the system remains in a state where a hot condition evaluates to false, this is considered a violation of the requirements.

2.4 Variables and Properties

Variables and properties are analogous to local method variables and data members in an object-oriented program. Both are finite discrete types, but they differ in lifetime and scope. *Lifetime* refers to the temporal duration in which a variable or property is available for use. A variable or property is *visible* if it can be legally referenced in a specific place in the requirements. *Scope* refers to the set of visible places.

Each LSC maintains a set of its own variables which are visible within the LSC. We assume that variables can appear only within assignments and conditions[3]. The lifetime of a variable begins when it is initialized to a value using an assignment and ends when the scenario terminates.

In contrast, properties are attributes of objects, not LSCs. An object's property is visible from any LSC in which the object appears. All properties have an infinite lifetime which commences at the start of execution. That is, the value of every property persists beyond the termination of any one scenario. One consequent of this setup is that property mutation in one LSC may alter the outcome of behaviors in another LSC

[3]In the full LSC language, variables can be used as message parameters. If the same message appears in more than one LSC, unifi cation is necessary to reconcile the variable bindings.

of the same requirements. Like variables, properties appear only in assignments and conditions.

In the full LSC definitions, variables can be assigned a constant value, the value of an object's property, or a user-supplied function. In this work, we assume that variables are assigned constant expressions only. This is a simplifying assumption only, as the other two cases could be handled similarly. The main requirement is that variables are assigned a value which is within their respective finite domain D. In the case of a user-supplied function, we further require the function to be defined for all inputs and range over any subset of D.

2.5 Events

Reactive systems produce behaviors that are visible to the outside world. LSCs observe these behaviors and update their state accordingly. Events serve as the connection between the behaviors of the reactive system and the LSC requirements. Events identify behaviors (e.g., a message name) and the objects participating in the behaviors (e.g., the sender and receiver). LSCs respond to events and map them to locations. A single event can map to one or more multiple locations. There are two possible reasons for the latter. The first is that several instances can participate in the same event—for example, instances synchronizing on a condition. The second is that the event needs to happen more than once during the scenario.

We proceed to define events in the following way. The set of LSC *events* is denoted by E, whose membership is comprised of the union of disjoint subsets E_v (*visible* *events*), E_h (*hidden events*), and E_u (*unrestricted* events).

Visible events are observable behaviors, which are comprised of the sending and receiving of messages between objects. The set of visible events is given by $E_v = E_v^1 \cup E_v^2 \cup E_v^3$, where E_v^1, E_v^2, E_v^3 are defined as follows. The set of synchronous message events is defined by $E_v^1 = \{(m.\texttt{name}, m.i_s.O, m.i_r.O) \mid m \in M \wedge m.\texttt{mode} = s\}$. That is, the name, sending object and receiving object for all synchronous messages. The sending and receiving are considered separate for asynchronous messages: $E_v^2 = \{(m.\texttt{name}, m.i_s.O) \mid m \in M \wedge m.\texttt{mode} = a\}$ and $E_v^3 = \{(m.\texttt{name}, m.i_r.O) \mid m \in M \wedge m.\texttt{mode} = a\}$.

The set of hidden events is given by $E_h = E_h^1 \cup E_h^2 \cup E_h^3$, which are defined as follows. The set of hidden events is $E_h^1 = \{pbegin, pend\}$, referring to the synchronization of instances at the beginning and end of a prechart, respectively. For assignments, we have $E_h^2 = \{a.\texttt{expr} \mid a \in A\}$ and for conditions, $E_h^3 = \{c.\texttt{expr} \mid c \in C\}$.

Unrestricted events with respect to an LSC are those which are irrelevant to the LSC scenario but may legally occur in any order[4] during its execution. This form of abstraction permits the user to focus on relevant behaviors while not necessarily excluding irrelevant ones. The set of unrestricted events, E_u, is specified a priori by the user and must satisfy $E_u \cap E_h \cap E_v = \emptyset$.

Function $evnts$ maps locations to events and satisfies the following:

- Assignments: $\forall a \in A\ \forall \ell, \ell' \in a.\texttt{locs} : evnts(\ell) = evnts(\ell')$
- Conditions: $\forall c \in C\ \forall \ell, \ell' \in c.\texttt{locs} : evnts(\ell) = evnts(\ell')$

[4]If there exist multiple LSCs in the requirements, the other LSCs could impose an ordering requirement.

- Synchronous Messages:

  $\forall m \in M : [m.\texttt{mode} = s] \Rightarrow \mathit{evnts}(m.\ell_s) = \mathit{evnts}(m.\ell_r)$

The first two statements tell us that all locations of an assignment or condition correspond to the same event. The third statement says that the sending and receiving of a *synchronous* message refers to the same event. We also require:

- Prechart Initiation:

  $\forall \ell, \ell' \in \{\min(\texttt{locs}(i) \cap Pch) \mid i \in I\} : \mathit{evnts}(\ell) = \mathit{evnts}(\ell') = \mathit{pbegin}$

- Prechart Completion:

  $\forall \ell, \ell' \in \{\max(\texttt{locs}(i) \cap Pch) \mid i \in I\} : \mathit{evnts}(\ell) = \mathit{evnts}(\ell') = \mathit{pend}$

- Main Chart Completion:

  $\forall \ell, \ell' \in \{\max(\texttt{locs}(i)) \mid i \in I\} : \mathit{evnts}(\ell) = \mathit{evnts}(\ell') = \mathit{pbegin}$

We refer to the bulleted items above as *synchronizing* events, since their main function is to synchronize all instances at particular points during execution. Specifically, the first two items state that the instances synchronize at the top and bottom of the prechart. All locations at the top and bottom of the prechart refer to the events pbegin and pend, respectively. A similar synchronization occurs at the bottom-most locations of the main chart. Notice that the top-most locations of the prechart and bottom-most locations of the main chart both correspond to the same event, pbegin. Consequently, control returns to the prechart at the precise instant that the main chart scenario completed.

2.6 Temporal Event Ordering

We now establish a series of assertions characterizing the temporal order of events based on the constructs appearing on an LSC.

For LSC L, we write ℓ_x^i to represent location x of instance i. Additionally, we write $evnts(\ell_x^i) <_T evnts(\ell_y^i)$ to express that $evnts(\ell_x^i)$ occurs strictly before $evnts(\ell_y^i)$. For some instances i, j and some $x \in \texttt{locs}(i), y \in \texttt{locs}(j)$, we write $evnts(\ell_x^i) =_T evnts(\ell_y^j)$ to express that $evnts(\ell_x^i)$ occurs at the same instant as $evnts(\ell_y^j)$. The temporal relation has a transitivity property which carries over to multiple instances in the sense that $evnts(\ell_x^i) <_T evnts(\ell_y^i)$ and $evnts(\ell_y^i) =_T evnts(\ell_x^j)$ implies $evnts(\ell_x^i) <_T evnts(\ell_x^j)$.

A *simultaneous region* is a maximal set of locations satisfying the temporal property "happens at the same time." More formally, a simultaneous region is a set of locations S such that there exists an event e where for all $\ell \in S$ we have $evnts(\ell) = e$. Moreover, for all locations $\ell' \in \texttt{locs}(L) \setminus S$, we have $evnts(\ell') \neq e$. We denote by $Sims_L$ the set of simultaneous regions of L.

2.7 Visible Events

Recall that visible events refer to the sending and receiving of messages. We impose the following requirements on visible events:

- $\forall m \in M : (m.\texttt{mode} = synchronous) \Rightarrow evnts(m.\ell_s) =_T evnts(m.\ell_r)$

- $\forall m \in M : (m.\texttt{mode} = asynchronous) \Rightarrow evnts(m.\ell_s) <_T evnts(m.\ell_r)$

The first requirement states that synchronous messages must be sent and received at the same instant. The second states that asynchronous message must be sent strictly before it is received.

2.8 The LSC Language

An LSC operates by continuously observing events and updating its state accordingly. We now define the notion of state more precisely.

2.8.1 Cuts

A *cut* is a tuple consisting of:

- A valuation mapping every object property to a value.
- A valuation mapping every LSC variable to a value.
- A mapping from every instance to a location.

Formally, the set of cuts is given by:

$$\mathit{Cuts} = \Sigma[\mathcal{O}] \times \Sigma[V] \times \prod_{i \in I} \texttt{locs}(i)$$

For a cut α, we write $\Sigma[\mathcal{O}](\alpha)$ to represent a specific valuation which maps every property to some value. Similarly, we write $\Sigma[V](\alpha)$ to represent the valuation mapping all LSC variables to a value in α. Finally, we write $\pi(\alpha)$ to represent the locations of the cut—an instance-to-location mapping—which due to the mutual distinction among locations, we express here for convenience as a *set* of locations. That is, $\pi(\alpha) = \{\ell_1, \ldots, \ell_k\}$, for some cut α where $k = |I|$.

The *initial cut*, denoted $\mathit{init}(\alpha)$, is the cut α where all instance locations are set to their top-most locations, all LSC variables are undefined ($\alpha(v) = \perp$, for all $v \in V$), and all object properties map to some initial value ($\alpha(p) = p.\Theta$, for all $p \in \mathit{Props}$).

The *temperature* of a cut α is a function of $\pi(\alpha)$. Specifically,

- $\mathit{temp}(\alpha) = H \Leftrightarrow \exists \ell \in \pi(\alpha) : \mathit{temp}(\ell) = H$
- $\mathit{temp}(\alpha) = W \Leftrightarrow \mathit{temp}(\alpha) \neq H \wedge \exists \ell \in \pi(\alpha) : \mathit{temp}(\ell) = W$
- $\mathit{temp}(\alpha) = C \Leftrightarrow \mathit{temp}(\alpha) \neq H \wedge \mathit{temp}(\alpha) \neq W$

2.8.2 Enablement

The *down-closure* of a cut α, denoted $\downarrow \alpha$ is the set of locations at and above the cut on the LSC. That is, $\downarrow \alpha = \{\ell^i_x \mid \ell^i_x \in \alpha \Rightarrow \forall y \leq x, \ell^i_y \in \alpha\}$. Conversely, the *top-line* of a set N, denoted $\top N$ is given by $\{\ell^i_x \mid \ell^i_x \in N \wedge \nexists \ell^i_y \in N \text{ s.t. } y > x\}$.

An event e is *enabled* from cut α, denoted $\mathit{enabled}(\alpha, e)$, iff there exists a simultaneous region s such that all of the following hold:

1. $\forall \ell \in s : \mathit{evnts}(\ell) = e$
2. $\forall \ell_j \in s : \ell_{j-1} \in \alpha$

3. $\forall m \in M : [m.\mathtt{mode} = a \wedge m.\ell_r \in s] \Rightarrow m.\ell_s \in \ \downarrow \alpha$

The first statement says that all locations of the simultaneous region refer to the same event e. The second statement requires that the cut be precisely one location above each of the locations of s. The third statement states, in the case of asynchronous messages, that the message must have been sent at or before cut α. We can also state explicitly that α' is the enabled cut with the 3-parameter assertion $\mathit{enabled}(\alpha, \alpha', e)$.

Enablement may also be expressed in terms of sim-regions. Let $S \subseteq \mathit{Sims}_L$ and define $\mathtt{locs}(S) = \bigcup_{s \in S} \mathtt{locs}(s)$. Given some subset $S \subseteq \mathit{Sims}_L$ and a simultaneous region s', we say that s' is enabled from S iff $\mathit{enabled}(\top\mathtt{locs}(S), \top(\mathtt{locs}(S) \cup \{s'\}), g(s'))$, written $\mathit{simenabled}(S, s')$.

2.8.3 Assignments and Conditions

An assignment $a \in A$ consists of an expression $\mathtt{expr}$ of the form "$\varphi := x$" where φ is an LSC variable or object property in $V \cup \mathit{Props}$ and x is a constant. We denote the *valuation* of φ in cut α by $\alpha(\varphi)$. We define an assertion over $\Sigma[\mathcal{O}]$ and $\Sigma[V]$, which relates two cuts α and α' with respect to an assignment:

$$\mathit{assign}(\alpha, \alpha', \varphi, x) = [\alpha'(\varphi) = x] \wedge [\forall \varrho \neq \varphi : \alpha'(\varrho) = \alpha(\varrho)]$$

This states that $\varphi = x$ in cut α' and the valuation of all other variables is unchanged between cuts α and α'. Similarly, a condition $c \in C$ consists of an expression $\mathtt{expr}$ of the form "$\varphi \# x$" where $\# \in \{=, \neq, <, >, \leq, \geq\}$. We introduce predicate eval :

$C \times \mathit{Cuts} \mapsto \mathbb{B}$ which, for expressions of the form "$\varphi \# x$," is defined by:

$$eval(c, \alpha) = c.\varphi \in V \Rightarrow [\alpha(c.\varphi) \neq \bot \ \wedge \ \alpha(c.\varphi)\ c.\#\ c.x]$$

Conditions are assigned a hot or cold temperature according to the function $temp$. Semantically, when an LSC arrives on a cold condition which evaluates to false, the LSC closes without violation. If it evaluates to true, chart progress is permitted to continue. A hot condition must evaluate to true before the chart may progress further, otherwise the LSC remains at the hot condition and will violate the requirements if it never progresses. We require the following for the next section:

$$unch(\alpha, \alpha', \varphi) : \alpha(\varphi) = \alpha'(\varphi)$$

This states that the value of φ remains unchanged between cuts α and α'.

2.8.4 Successor Cuts

Let α, α' be cuts and let $\alpha_\Delta = \pi(\alpha') \setminus \pi(\alpha)$. That is, the set of locations of cut α' minus the locations of α. Intuitively, this refers to the locations of the event that caused the cut to change. Transitions between cuts are defined by a total function $\mathit{Next} : \mathit{Cuts} \times E \mapsto 2^{\mathit{Cuts}}$. We say α' is a *successor* of α iff there exists e such that $\alpha' = \mathit{Next}(\alpha, e)$. This is the case whenever all of the following hold:

1. $\mathit{enabled}(\alpha, \alpha', e) \ \vee \ \mathit{init}(\alpha')$

   Event e is enabled from α, or leads to the initial cut.

2. $\forall a \in A : [\texttt{locs}(a) \subseteq \alpha_\Delta] \Rightarrow [assign(\alpha, \alpha', a.\varphi, a.x)]$

   When arriving on a variable assignment, α' will reflect the new value of variable or object property φ.

3. $\forall \varphi \in (V \cup Props) : \; [[\forall a \in A : a.\varphi \neq \varphi \; \vee \; \texttt{locs}(a) \nsubseteq \alpha_\Delta] \; \wedge \; \neg init(\alpha') \Rightarrow unch(\alpha, \alpha', \varphi)]$

   If not arriving on an assignment or returning to the initial state, all variable and property valuations remain unchanged.

4. $\forall c \in C : [\texttt{locs}(c) \subseteq \alpha] \Rightarrow eval(\alpha, c) = \text{TRUE}$

   Advancing beyond a condition in the LSC requires that the condition evaluate to true.

5. $[temp(\alpha') = C] \Leftrightarrow init(\alpha')$

   Transitions into cold cuts lead into the initial state (cause the LSC to close).

6. $\forall c \in C : [\texttt{locs}(c) \subseteq \alpha \; \wedge \; temp(c) = C \; \wedge \; eval(\alpha, c) = \text{FALSE}] \Rightarrow init(\alpha')$

   Transitions into false cold conditions lead into the initial state (cause the LSC to close).

7. $init(\alpha') \Rightarrow \forall v \in V : \alpha'(v) = \bot$

   All LSC variables are undefined in the initial state.

2.8.5 Runs & Computations

An *LSC run* is an infinite sequence of cuts $\pi = \alpha_1, \alpha_2, \ldots$, satisfying:

- $init(\alpha_1)$ (initiality)

- $\forall i \, \exists e_i : \alpha_{i+1} \in \mathit{Next}(\alpha_i, e_i)$ (consecution)

Let $\mathit{Accept} = \{\alpha \mid \mathit{temp}(\alpha) \neq H\}$. Let $\mathit{Inf}(\pi)$ be the set of cuts occurring infinitely often in run π. An *LSC computation* is an LSC run also satisfying: $\mathit{Accept} \cap \mathit{Inf}(\pi) \neq \emptyset$. An *event trace* is the sequence of events induced by a computation. The *LSC language* of an LSC is the set of all event traces. If $\sigma = e_i, \ldots, e_k$ is a subsequence of an event trace, we write $\mathit{Next}^*(\sigma)$ iff there exists a sequence of cuts $\alpha_i, \ldots, \alpha_{k+1}$ such that $\alpha_{j+1} = \mathit{Next}(\alpha_j, e_i)$ for all $i \leq j \leq k$.

2.8.6 Hidden Events

We distinguish between event behaviors which are produced by the reactive system (i.e., messages) and those which are internal to the LSC (i.e., assignments, conditions, and synchronization points). Later, we will construct an automaton which monitors a sequence of visible behaviors and decides whether it satisfies or violates the LSC requirements. It is first necessary to establish a semantics that determines precisely how hidden events are to be carried out based solely on the occurrences of visible events. We present one particular interpretation which will be assumed for the purposes of our construction in Chapter 4.

Our interpretation requires hidden events to execute at the instant they become enabled. This means, for example, that if an assignment or condition appears directly below a message on an instance line, it would execute at the same instant as the message. If the assignment or condition is anchored to other instances, it must also be enabled with respect to those instances. Otherwise, it executes upon the occurrence of the first visible event to enable it. Subsequences of consecutive enabled hidden events execute at the same time instant provided each is enabled. If differing linearizations of these hidden event subsequences exist, the LSC chooses one arbitrarily. The chosen linearization could be significant if multiple assignments to the same variable or property exist in the subsequence.

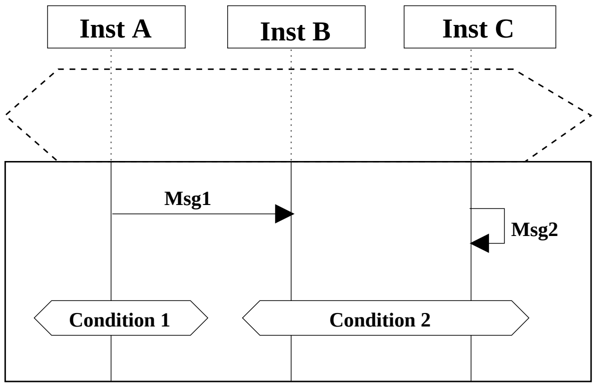


Figure 2.3: Hidden Event Generation

To illustrate, consider Fig. 2.3 which depicts two messages and two conditions. We omit the conditional expressions for this example. We first consider the events which occur on instance A only. Since message 1 immediately precedes condition 1 on instance A, it is the case that condition 1 will occur at the same instant as message 1. Condition 2, on the other hand, may only be evaluated once both messages have occurred since it is anchored to another instance. Therefore, condition 2 will be evaluated at the same instant as whichever message occurs later in time. Moreover, if message 2 occurs before message 1, then both conditions will be evaluated at the same instant as message 1.

Let $e_1, e_2, \ldots$ be the event trace induced by some LSC run. We impose the following temporal requirements:

- $\forall i : e_i \in (E_v \cup E_h) \wedge e_{i+1} \in E_h \Rightarrow e_i =_T e_{i+1}$

- $\forall i : e_i \in (E_v \cup E_h) \wedge e_{i+1} \in E_v \Rightarrow e_i <_T e_{i+1}$

2.8.7 Chart Satisfaction and Violation

A sequence of events $\sigma = e_1, \ldots, e_n$ *satisfies the prechart* of a Universal LSC (denoted $\sigma \models Pch$) if $e_1 = pbegin$, $e_n = pend$ and $Next^*(\sigma)$ holds. Intuitively, σ satisfies the prechart if it terminates with a satisfying subsequence and causes the bottom-most locations of the prechart to be reached.

We say that $\sigma = e_1, \ldots, e_n$ *violates the prechart* (denoted $\sigma \not\models Pch$) if $e_1 = pbegin$, $e_n = pbegin$, $Next^*(\sigma)$, and $e_i \neq pend$ for $1 < i < n$. We say σ is a *minimal violating sequence* if there does not exist j such that $e_j = pbegin$ for $1 < j < n$.

An event sequence $\sigma = e_1, \ldots, e_j, \ldots, e_n$ *satisfies the main chart* of a universal LSC L (denoted $\sigma \models L$) if $[e_1, \ldots, e_j] \models Pch$, $e_n = pbegin$ and $Next^*(\sigma)$ holds.

Sequence $\sigma = e_1, e_2, \ldots$ *violates the main chart*, and therefore the requirements, if there is a subsequence $\sigma_j = [e_1 \ldots, e_j]$ such that $\sigma_j \models Pch$ and either $\nexists k > j$ s.t. $e_k = pbegin$ or $\neg Next^*(\sigma)$ holds.

2.9 Example

To familiarize the unacquainted reader, we present the example of Fig. 2.4. The scenario depicts communication between a user, buffered keypad, and CPU. The prechart states that whenever the user sends $PressOn$ and then $PressKey$, the interaction appearing in the main chart must follow thereafter. Namely, that `Keypad` sends the $ProcessBuf$

message to `CPU`, followed by the `CPU` sending *ClearBuf* to `Keypad`, followed finally by the `CPU` sending *Notify* to `User`.

The example is adorned with horizontal dashed lines to show three points of synchronization, each annotated by the name of the corresponding hidden event. All scenarios begin with the instances synchronized to the top-most location of each instance. The hidden event which describes the act of synchronization is $pbegin$. All instances synchronize again at the bottom of the prechart, where event $pend$ occurs. In the main chart, all instances synchronize at the bottom-most locations of each instance upon the occurrence of $pbegin$. Note that this is a continuation point which signifies a return of control to the prechart.

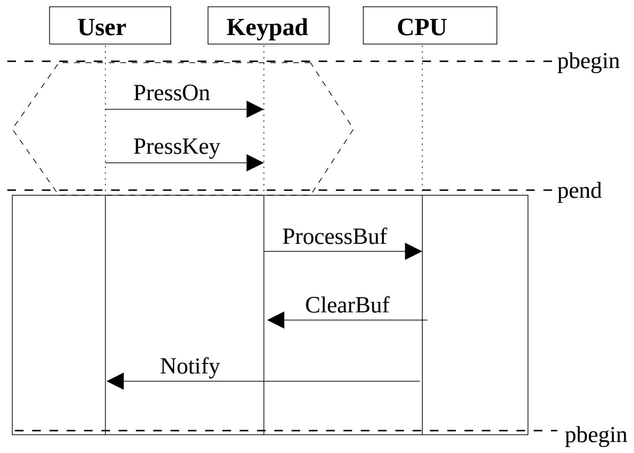


Figure 2.4: Fairness Free Universal LSC

We now consider various event sequences and differentiate satisfying behaviors from violating behaviors. For brevity, we omit the identity of the sender and receiver instances of a message (e.g., the message *ClearBuf* is understood by context to be sent by `CPU` and received by `Keypad`). All messages in this example are synchronous, so we use the name of each message to denote the simultaneous sending and receiving of the message. Our discussions consider finite fragments of LSC computations.

We begin by considering visible events only. Consider the event sequences $\sigma_1 = \mathit{PressOn}, \mathit{PressKey}$ and $\sigma_2 = \mathit{PressKey}$. Sequence σ_1 satisfies the prechart whereas σ_2 violates the prechart. However, $(\sigma_1)^\omega$ is not in the language, whereas $(\sigma_2)^\omega$ is. This is because σ_1 generates the obligation that the subsequence which follows must satisfy the main chart (which it doesn't), whereas the prechart violating behavior of σ_2 never creates any obligation.

Let $\sigma_3 = \mathit{PressOn}, \mathit{PressKey}, \mathit{ProcessBuf}, \mathit{ClearBuf}, \mathit{Notify}$. This sequence satisfies the requirements since it satisfies the prechart and main chart. Let $\sigma_4 = \mathit{PressOn}, \mathit{PressKey}, \mathit{ProcessBuf}, \mathit{Notify}$. Sequence σ_4 violates the requirements because the prechart is satisfied but $\mathit{ProcessBuf}$ is not followed by $\mathit{ClearBuf}$ as required by the main chart.

Until now, we omitted hidden events from the discussion of Fig. 2.4. Let us assume that an object system outputs the visible event sequence σ_3. From this, the LSC carries out the sequence $\mathit{pbegin} <_T \mathit{PressOn} <_T \mathit{PressKey} =_T \mathit{pend} <_T \mathit{ProcessBuf} <_T \mathit{ClearBuf} <_T \mathit{Notify} =_T \mathit{prebegin}$. All computations begin with the hidden event pbegin. Hidden events are executed as soon as they become enabled, and each occurs at the same time instant as the immediately preceding visible event. For example, pend occurs next in the order and at the same temporal location as $\mathit{PressKey}$. Likewise, pbegin immediately follows Notify.

2.10 Multiple LSC Requirements

We now consider LSC requirements consisting of more than one LSC. Traditional methods of scenario composition, such as that seen in MSCs [MSC99], offer the choice of executing multiple scenarios sequentially, alternatively, or concurrently. The choices must be stated up-front by the user using high-level MSCs (hMSCs). In the realm of multiple-LSC requirements, concurrency is dynamic in the sense that a particular set of scenarios may run concurrently, sequentially, alternatively, or any combination thereof, based not on fixed user input but rather on the input behaviors. The dynamic behavioral approach to LSC composition gives rise to the possibility of interesting interactions between several executing scenarios.

Multiple LSC requirements share the same set of objects, and therefore the same object properties, but have distinct sets of LSC variables. For this reason, multiple-LSC requirements cannot be expressed simply as the product of cuts from each LSC, otherwise duplication of object properties occurs.

Let $\Gamma_j = L_j.\Sigma[V] \times L_j.\prod_{i \in I} \texttt{locs}(i)$ represent the set of LSC variable valuations and the instance-to-location mapping for LSC j only. Given an object system $\mathcal{O}$ and LSCs $L_1, \ldots, L_n$, we define *multiple LSC requirements* by $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ where each L_i is an LSC. An *m-cut* (multi-cut) is the tuple:

$$MCuts = \Sigma[\mathcal{O}] \times \Gamma_1 \times \cdots \times \Gamma_n$$

In the case of single LSCs, the assertions $assign, unch$ and predicate $eval$ are defined over the valuations in $\Sigma[\mathcal{O}]$ and $\Sigma[V]$ for object properties and variables, respectively. For multiple LSCs, there exists only one set of valuations $\Sigma[\mathcal{O}]$ but one $\Sigma[V]$ valuation for each LSC j—each referenced by Γ_j. It is understood that when assignments or conditions over an LSC variable occur in LSC i, the semantics of $assign$, $eval$ and $unch$ are carried out with respect to Γ_i. This is assumed to be the case from this point forward.

We can project an m-cut onto an ordinary single LSC cut with the function $SCut : MCuts \times \mathbb{N} \mapsto 2^{Cuts}$. We write $SCut(\beta, i) = \alpha$ to denote the cut α for LSC i in m-cut β. Each single LSC cut has its own successor function $Next$, as we have previously described. We refer to the particular successor function of LSC i with $Next_i$.

The *initial m-cut* is the m-cut where $init(SCut(\beta, i))$ holds for all i. We write $init(\beta)$ to assert that m-cut β is initial.

A function $MNext : MCuts \times E \mapsto MCuts$ is defined in terms of original cuts. Specifically, $\beta' = MNext(\beta, e)$ iff there exists e such that for all $i : SCut(\beta', i) = Next_i(SCut(\beta, i), e)$. We say an m-cut β' is a *successor* of β whenever $MNext(\beta, e) = \beta'$ holds for some event e.

In the spirit of single LSCs, a *run* of $\mathcal{L}$ is an infinite sequence of m-cuts $\pi = \beta_1, \beta_2, \ldots$, satisfying:

- $init(\beta_1)$ (initiality)
- $\forall i \, \exists e_i : \beta_{i+1} \in MNext(\beta_i, e_i)$ (consecution)

A multiple-LSC *computation* is a run satisfying $Accept_i \cap Inf(\pi) \neq \emptyset$ for all i, where $Accept_i = \{\alpha \mid temp(SCut(\beta, i)) \neq H\}$. An *event trace* is the sequence of events induced by a computation. The *language* of an LSC is the set of all event traces.

2.11 Self-Spawning LSCs

It is possible for a single universal LSC to spawn and execute several instantiations of itself concurrently—a phenomenon we refer to as *self-spawning scenarios*. This happens, for example, if the LSC is configured to monitor every input for a *minimal event*—an event which causes the prechart to progress from the top-most locations. Several "copies" of the LSC could therefore exist, each monitoring the input event sequence starting from differing points in the trace.

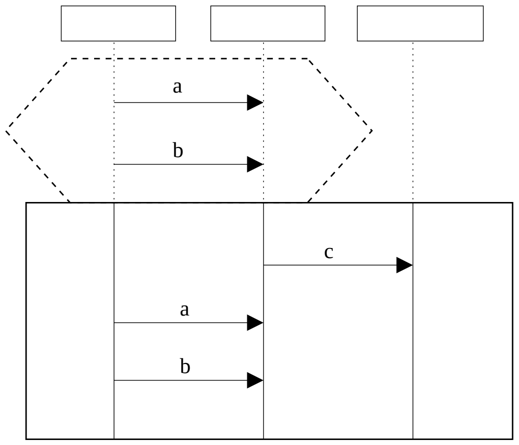


Figure 2.5: Self-Spawning LSC

Consider the LSC requirements of Fig. 2.5, consisting of one LSC. When the messages a, b occur, the LSC requires c, a, b to occur. By carrying out the required behaviors of the main chart, however, it is possible to satisfy the prechart. This gives rise to a requirement for an infinite cycle of behaviors, namely $(abc)^\omega$, anytime the prefix ab occurs.

We can proceed in one of two directions. The first is to monitor for prechart activation only when the chart is at the initial location. This approach would essentially ignore the prechart whenever the LSC is at a location other than the initial. The second approach is to monitor the prechart at all stages of the execution and permit requirements such as the one in Fig. 2.5.

In this work, we choose the former approach in which scenarios cannot self-spawn in this way. That is, we assume one “copy” of each LSC can exist. The main reason for doing this is so we may restrict our attention to a simpler subset of features. Self-spawning scenarios are capable of producing behaviors which could require special handling. For the interested reader, self-spawning scenarios are considered in [PGZ05].

Chapter 3

Timed LSC Requirements

3.1 Introduction to Time

Many software designs, aside from specifying legal orderings of behaviors, also include timing constraints. Using the previously introduced LSC assignment and condition constructs along with special time variables, LSCs can express timing constraints. We refer to LSCs with such capability as *timed LSCs*. Timing constraints are generally used to impose upper and lower time bounds on a certain sequence of events, which has the effect of reducing the number of acceptable event traces. Event sequences which are acceptable without timing constraints could become unacceptable if the sequences do not occur within the prescribed time bounds.

In this chapter, we consider two particular models of timed LSCs which shall be translated to formal models in later sections. We first consider the simpler case of *discrete-time* in which time is assumed to be in the domain of natural numbers. Without loss of generality, we further assume that time deltas (i.e., consecutive time units) are separated by a time value of 1. We do not concern ourselves with the specific interpretation of time units; that is, whether the units express minutes, seconds, etc.

We follow up the discrete time discussion by considering *dense-time* LSCs, in which time units are monotonically increasing values expressed over the domain of non-negative real numbers. Like before, we are not interested in the particular interpretation of time units. However, one important distinction between the two models is that dense-time does not assume any fixed time delta.

3.2 Related Work

Although several LSC language extensions have been proposed since the seminal LSC paper [DH01], literature on timed LSCs, in particular, has been relatively scant. The first proposal for LSC time constraints to our knowledge was offered in [KW01]. Another scheme, which more closely follows the spirit of the untimed LSC variant, was introduced a year later in [HM02]—our translation most closely follows this approach. The treatment of timed LSCs presented in this chapter is based on the author's contributions to the work in [PGZ05].

3.3 Single Discrete-Time LSCs

We consider LSC requirements which can express time in discrete units. For this purpose, we incorporate a global *system clock*, ranging over the naturals, into our subset of the LSC language. The system clock is viewed as a special object `Clock` with a message $Tick$. Execution of $Tick$ causes the clock value to increment by 1. The current value of the system clock is depicted on the LSC as *Time*.

All behaviors described by an untimed LSC can be interleaved with occurrences of $Tick$ in a timed LSC. Assignments and conditions similar to those seen in the previous chapter can be used to store *Time* into a special *time-stamp variable* (ts-var) and later test time variables with respect to *Time*. Time-stamp variables range over $\mathbb{N} \cup \{\perp\}$ where $\perp$ is undefined.

Storage of the system clock value into a ts-var is accomplished exclusively using special LSC time assignments of the form "$\varphi :=$ *Time*," where φ is a ts-var. Only *Time* may be stored into a ts-var—constants, ordinary LSC variable or object property r-values are not permitted to be stored in a ts-var. The value of a ts-var may be recalled using an LSC condition restricted to the form "*Time* $\# \ \varphi + const$" where $\# \in \{<, >, \leq, \geq, =, \neq\}$ and $const$ is a constant.

Like LSC variables, ts-vars are undefined at the beginning of each LSC execution. In contrast to LSC variables, however, the range of time-stamp variable values is theoretically unbounded. Consequently, discrete-time LSCs are infinite state systems.

3.3.1 Definition

We expand the definition of LSCs, as presented in section 2.2, to include ts-vars. Let $L = \langle I, M, V, V_T, A, C, Pch, evnts, temp \rangle$, where $I, M, V, A, C, Pch, evnts, temp$ are as before and:

- V_T—a set of time-stamp variables, each of which is defined by $\langle \texttt{name}, \omega \rangle$ where `name` is the name of the variable and $\omega \in \mathbb{N} \cup \{\perp\}$ is the value.

The set of ts-vars is defined separately from standard finite discrete LSC variables in order to distinguish between the unbounded ts-vars variables and the bounded LSC variables. As before, the assertions $eval$, $assign$, $unch$ are still applicable to assignments and conditions over the LSC variables and object properties. We will introduce new assertions for dealing with ts-vars shortly.

3.3.2 Timed Cuts

The set of *timed cuts* is given by:

$$\mathit{TCuts} = \mathit{Cuts} \times \Sigma[V_T]$$

where *Cuts* is the set of cuts as seen before and $\Sigma[V_T]$ is the set of ts-var valuations.

In future sections, we distinguish original cuts from timed cuts by referring to them as *untimed* or *timed* cuts, respectively. If τ is a timed cut, the *untimed cut of* τ, written $\mathit{Untime}(\tau)$, is the projection of *Cuts* on τ. We denote by $\tau(\varphi)$ the valuation of ts-var $\varphi \in V_T$. We use $\pi(\tau)$, as before, to project the element of the cut referring to the instance-to-location mapping location. The *initial timed cut* is given by $init(\tau) = \forall \varphi \in V_T : \tau(\varphi) = \perp \;\wedge\; init(\mathit{Untime}(\tau))$.

3.3.3 Discrete-Time Assertions

We characterize the semantics of LSC time assignments using the assertion $assign_t$. Given a ts-var assignment a, a timed cut τ and a time t, the assertion states that variable $a.\varphi$ takes on the value t in cut τ:

$$assign_t(a, \tau, t) : \tau(a.\varphi) = t$$

We provide a similar assertion, $eval_t$, for ts-var conditions:

$$eval_t(c, \tau, t) : \tau(c.\varphi) \neq \bot \ \wedge \ t \ c.\# \ \tau(c.\varphi) + c.x$$

where $\# \in \{=, \neq, <, >, \leq, \geq\}$. Finally,

$$unch_t(\tau, \tau', \varphi) : \tau(\varphi) = \tau'(\varphi)$$

3.3.4 Successor Cuts

Similar to section 2.8.4, we use the shortcut $\tau_\Delta = \pi(\tau') \setminus \pi(\tau)$, where τ_Δ is understood to refer to the simultaneous regions in τ' not appearing in τ. We say timed cut τ' over system clock value t is a *discrete-time successor* of τ if there exists an event $e \in E$ such that $\tau' \in TNext(\tau, e)$. This is the case iff all of the following hold:

1. $\mathit{Untime}(\tau') = Next(\mathit{Untime}(\tau), e)$

   The untimed LSC semantics of section 2.8.4 hold.

2. $\forall a \in A : [\texttt{locs}(a) \subseteq \tau_\Delta \ \wedge \ a.\varphi \in V_T] \Rightarrow assign_t(a, \tau', t)$

When arriving on a ts-var assignment, store t in ts-var $a.\varphi$.

3. $\forall \varphi \in V_T\ [[\forall a \in A : a.\varphi \neq \varphi\ \vee\ \texttt{locs}(a) \not\subseteq \tau_\Delta]\ \wedge\ \neg init(\tau') \Rightarrow unch_t(\tau, \tau', \varphi)]$

   If ts-var φ does not appear in an assignment, then its value is unchanged.

4. $\forall c \in C : [\texttt{locs}(c) \subseteq \tau\ \wedge\ c.\varphi \in V_T] \Rightarrow [eval_t(c, \tau, t) = \text{TRUE}]$

   Advancing from a condition requires that it evaluate to true.

5. $\forall c \in C : [\texttt{locs}(c) \subseteq \tau\ \wedge\ temp(c) = C\ \wedge\ eval(\tau, c) = \text{FALSE}] \Rightarrow init(\tau')$ Cold conditions which evaluate to false lead into the initial state.

6. $init(\tau') \Rightarrow \forall v \in V_T : \tau'(v) = \perp$

   All LSC ts-vars are undefined in the initial state.

3.3.5 Runs and Computations

Let $Accept = \{\tau \mid temp(\tau) \neq H\}$. A *discrete-time LSC run* is an infinite sequence of cuts $\pi = \tau_1, \tau_2, \ldots$, satisfying:

- $init(\tau_1)$ (initiality)
- $\forall i\ \exists e_i : \tau_{i+1} \in TNext(\tau_i, e_i)$ (consecution)

A *discrete-time LSC computation* is a discrete-time LSC run satisfying: $Accept \cap Inf(\pi) \neq \emptyset$. A *discrete-time event trace* is the sequence of events induced by a computation. The *discrete-time language* of an LSC is the set of all discrete-time event traces.

3.4 Single Dense-Time LSCs

Within the domain of dense time, values of the ts-vars and system clock both range over the non-negative reals ($\mathbb{R}_{\geq 0}$). A single dense-time LSC is defined similarly to the discrete-time case, where:

- V_T—a set of time-stamp variables, each of which is defined by $\langle \texttt{name}, \omega \rangle$ where `name` is the name of the variable and $\omega \in \mathbb{R}_{\geq 0} \cup \{\bot\}$ is the value.

As in the discrete-time case, the set $\Sigma[V_T]$ of ts-var valuations is infinite and unbounded. However, these valuations can be mapped into a finite number of *clock regions* according to [AD94]. The set of timed cuts is the same as in the discrete case, except substituting the new dense-time version of V_T:

$$\mathit{TCuts} = \mathit{Cuts} \times \Sigma[V_T]$$

where *Cuts* is the set of untimed cuts and $\Sigma[V_T]$ is defined as above. Note that we use the same name V_T to refer to the ts-vars for both the dense-time case here and in the previous section for the discrete-time case. This permits us to interchange the two models so as to better express the later results of this work more succinctly. For a timed cut τ, we again denote by $\pi(\tau)$ the set of locations belonging to the cut.

An instance `Clock` with message Tick is not used, as in the discrete-time case[1]. In our approach, the passage of time is expressed by associating each input event with

[1]Generally, it is possible to use a Tick message which is parameterized by a dense "real time" value. However, we do not consider parameterized messages in this work.

an explicit time value. We define a *timed event* by the 2-tuple $(E, \mathbb{R}_{\geq 0})$. Let $TEvents$ be the infinite set of timed events.

Assertions $eval$ and $assign$ are used for characterizing the semantics of LSC assignments and conditions over over LSC variables and object properties. That is, assignments and conditions not involving time constraints. In section 3.3.3, the discrete-time analogues $eval_t$ and $assign_t$ were presented. We now redefine these assertions for the dense-time case.

Assertion $assign_t$ takes three parameters: an LSC assignment $a \in A$, a timed cut $\tau \in$ *TCuts*, and a time value $t \in \mathbb{R}_{\geq 0}$. The latter parameter identifies the time at which the assignment itself is carried out.

$$assign_t(a, \tau, t) : a.\varphi \in V_T \Rightarrow \tau(a.\varphi) = t$$

Similarly, $eval_t$ also requires three parameters consisting of a condition $c \in C$, timed cut $\tau \in$ *TCuts*, and time value $t \in \mathbb{R}_{\geq 0}$.

$$eval_t(c, \tau, t) : c.\varphi \in V_T \Rightarrow [\tau(c.\varphi) \neq \bot \ \wedge \ t \ c.\# \ \tau(c.\varphi) + c.x]$$

Finally, $unch_t$ takes three parameters: two timed cuts τ, τ' and a ts-var φ. By $\tau(\textit{Time})$ we denote the system clock value in timed cut τ[2].

$$unch_t(\tau, \tau', \varphi) : \tau'(\varphi) = \tau(\varphi) + (\tau'(\textit{Time}) - \tau(\textit{Time}))$$

[2]This system clock value can be extracted from a timed cut by reserving a special ts-var and assigning it the system clock valuation.

3.4.1 Successor Cuts

We define a transition function $\mathit{TNext} : \mathit{TCuts} \times \mathit{TEvents} \mapsto 2^{\mathit{TNext}}$. Let τ, τ' be timed cuts and $\tau_\Delta = \pi(\tau') \setminus \pi(\tau)$. We say timed cut τ' over system clock value t is a *dense-time successor* of τ if there exists an event $e \in E$ and time $t \in \mathbb{R}_{\geq 0}$ such that $\tau' \in \mathit{TNext}(\tau, (e, t))$. This is the case iff all of the following hold:

1. $\mathit{Untime}(\tau') = \mathit{Next}(\mathit{Untime}(\tau), e)$

   Assertions over the untimed cuts.

2. $\forall a \in A : [\mathtt{locs}(a) \subseteq \tau_\Delta] \Rightarrow \mathit{assign}_t(a, \tau', t)$

   When arriving on an assignment for ts-var φ, the timer value of φ is reset to zero.

3. $\forall \varphi \in V_T \; [[\forall a \in A : a.\varphi \neq \varphi \; \vee \; \mathtt{locs}(a) \nsubseteq \tau_\Delta] \; \wedge \; \neg \mathit{init}(\tau') \Rightarrow \mathit{unch}_t(\tau, \tau', \varphi)]$

   If ts-var φ does not appear in an assignment, then its value is t.

4. $\forall c \in C : [\mathtt{locs}(c) \subseteq \tau_\Delta] \Rightarrow [\mathit{eval}_t(c, \tau, t) = \text{TRUE}]$

   Advancing beyond a condition in the LSC requires that the condition evaluate to true.

5. $\mathit{init}(\mathit{Untime}(\tau')) \Rightarrow [\forall v \in V_T : \tau'(v) = \bot]$

   All LSC ts-vars are undefined in the initial state.

3.4.2 Runs and Computations

A *dense-time LSC run* is a sequence of pairs $\tau = (\tau_1, t_1), (\tau_2, t_2), \ldots$ satisfying:

1. $\mathit{init}(\tau_1)$ (initiality)

2. $\forall i \; \exists e \in E : \tau_{i+1} \in TNext(\tau_i, (e_i, t_i))$ (consecution)

3. $\forall i, j : e_i, e_j \in E_v \; \wedge \; i < j \Rightarrow t_i < t_j$ (monotonicity)

4. $\forall t \in \mathbb{R}_{\geq 0} \; \exists i \geq 1$ s.t. $t_i > t$ (progress)

If $\tau = \tau_1, \tau_2, \ldots$ is a dense-time run, then we express the *untimed* projection of τ as $\mathit{Untime}(\tau) = \mathit{Untime}(\tau_1), \mathit{Untime}(\tau_2), \ldots$. A *dense-time LSC computation* is a dense-time LSC run satisfying $Accept \; \cup \; Inf(\tau)$, where $Inf(\tau)$ is the set of timed cuts appearing infinitely often in τ and $Accept = \{\mathit{Untime}(\tau) \mid temp(\mathit{Untime}(\tau)) \neq H\}$. A *dense-time event trace* is the sequence of events induced by a dense-time LSC computation. The *dense-time language* is the set of all dense-time event traces.

3.5 Multiple Dense-Time LSCs

We now present the formal definitions for dense-time multiple timed LSC requirements. A similar formulation can be applied to the untimed and discrete-time cases, but are not considered here.

Multiple timed LSC requirements collectively describe the behaviors of an object system. LSC variables and ts-vars are meta-variables used for the purpose of constraining the behaviors of the objects. Both are unique to each particular LSC and the valuations of both begin in an undefined state at the start of each scenario and become undefined again at the end. In contrast, object property valuations define the state of the object system throughout its entire lifetime. Furthermore, they persist indefinitely, beyond the lifetime of any one scenario. Accordingly, the variables, ts-vars, and the current position of each instance of an LSC are components that are unique to the LSC, whereas object properties are global to all LSCs in the requirements.

Given an object system $\mathcal{O}$ and LSCs $L_1, \ldots, L_n$, we define multiple timed LSC requirements by $\mathcal{L}_{\mathit{multi}} = \langle L_1, \ldots, L_n \rangle$ where each L_i is an LSC as defined previously.

The *state* of LSC j is given by: $\Gamma_j = L_j.\Sigma[V] \times L_j.\prod_{i \in I} \texttt{locs}(i) \times L_j.\Sigma[V_T]$. This represents the set of variable valuations, instance-location mapping, and ts-vars, respectively, for LSC L_j. The set of ts-vars is assumed to operate in the dense time domain for the purpose of this work, but a discrete time domain could also be adopted.

A *timed m-cut* (multi-cut) is a tuple:

$$\mathit{TMCuts} = \Sigma[\mathcal{O}] \times \Gamma_1 \times \cdots \times \Gamma_n$$

A projection function $\mathit{TSCut} : \mathit{TMCuts} \times \{1, \ldots, n\} \mapsto \mathit{TMCuts}$ yields a timed cut given the unique number of a dense-time LSC. A timed m-cut β is an *initial timed m-cut*, denoted $\mathit{init}(\beta)$ iff $\mathit{init}(\mathit{TSCut}(\beta, i))$ holds for $1 \leq i \leq n$.

A visible event of the composed system is defined by $E_v = \bigcup_i L_i.E_v$. For some LSC j, we assume that if $e \in E_v$ and $e \notin L_j.E_v$ then $e \in L_j.E_u$.

We define $\mathit{TMNext} : \mathit{TMCuts} \times (E, \mathbb{R}_{\geq 0}) \mapsto 2^{\mathit{TMCuts}}$ as the transition function of the composed requirements. Let TMNext_i refer to the successor function of LSC i. We say that β' is a *timed m-cut successor* of β, written $\beta' \in \mathit{TMNext}(\beta, (e, t))$, if there exists event e and time t such that:

1. $e \in E_v \Rightarrow \forall i : \mathit{TSCut}(\beta', i) \in \mathit{TMNext}_i(\mathit{TSCut}(\beta, i), (e, t))$

2. $\exists i : e \in L_i.E_h \Rightarrow \mathit{TSCut}(\beta', i) \in \mathit{TMNext}_i(\mathit{TSCut}(\beta, i), (e, t))$

Intuitively, visible events cause a synchronous transition to be taken among all LSCs, since each is expected to respond in some way to the occurrence of the event. Each hidden event, on the other hand, represents an internal behavior for just one particular LSC. The case distinction is necessary since LSCs only recognize their own hidden events, otherwise the semantics could be entirely synchronous.

3.5.1 Runs and Computations

A *dense-time run* of $\mathcal{L}_{multi}$ is a pair $(\mathit{TMCuts}, \mathbb{R}_{\geq 0}) = (\beta_1, \nu_1), (\beta_2, \nu_2), \ldots$ satisfying:

1. $\mathit{init}(\beta_1)$ (initiality)

2. $\forall i\ \exists e_i\ \exists t_i : \beta_{i+1} \in \mathit{TMNext}(\beta_i, (e_i, t_i))$ (consecution)

3. $\forall i, j : e_i, e_j \in E_v\ \wedge\ i < j \Rightarrow \nu_i < \nu_j$ (monotonicity)

4. $\forall t \in \mathbb{R}_{\geq 0}\ \exists i : \nu_i > t$ (progress)

A *dense-time computation of* $\mathcal{L}_{multi}$ is a dense-time run of $\mathcal{L}_{multi}$ which satisfies, for all i, $\mathit{Accept}_i\ \cap\ \mathit{Inf}(\pi) \neq \emptyset$ where $\mathit{Accept}_i = \{\beta \mid \mathit{temp}(\mathit{TSCut}(\beta, i)) \neq H\}$. An *timed event trace* of $\mathcal{L}_{multi}$ is the sequence of events induced by a timed computation and the *timed language* of $\mathcal{L}_{multi}$ is the set of all timed event traces.

3.6 Timed LSC Example

We now consider a four-wheeled robotic agent that navigates the perimeter of a round-cornered rectangle drawn in black on a white background. The agent is controlled by a microprocessor with a single light sensor input and an output that controls motor speed for each wheel. Differential wheel power is used to turn the agent left or right. We assume that the agent is initially placed directly over any edge of the rectangle, positioned to travel clockwise as shown in Fig. 3.1.

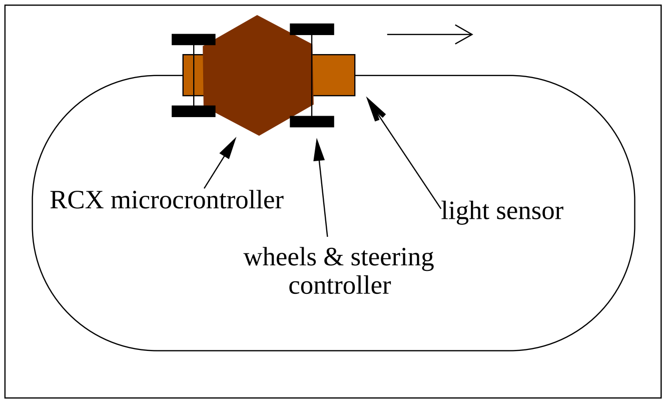


Figure 3.1: Robot Test Pad

To make the LSC more readable, we introduce a special up/down arrow notation on the left side of assignments and conditions to indicate either the preceding (up arrow) or succeeding (down arrow) visible event with which each is associated. The same requirements can be expressed without the use of this notation.

In Fig. 3.3, variable T is assigned at the same time instant that message *TurnRight* is sent, and the condition over $T2$ executes at the same time instant that *TurnRight* is received.

The first scenario of this example (Fig. 3.2) describes what should happen when

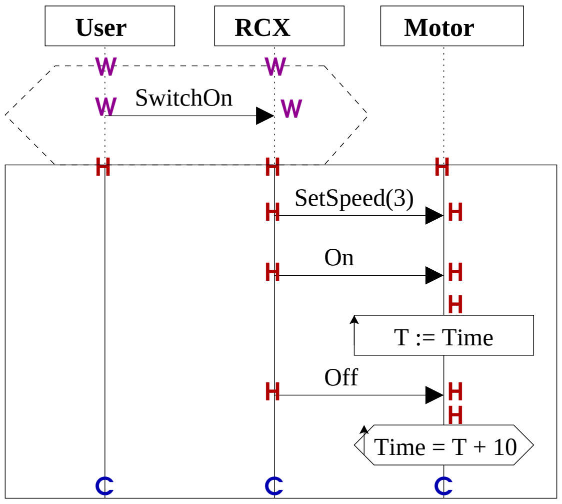


Figure 3.2: Scenario 1: Switching Robot On

the user switches on the micro controller. In particular, the motor speed is set to 3 and the micro controller switches on the motor. Precisely ten time units after the motor is switched on, the motor is switched off by the micro controller.

The second scenario (Fig. 3.3) involves a description of how the agent will navigate the black line. This is carried out by means of a light sensor which is connected to the micro controller. The light sensor can observe two possible conditions: bright or dark. When the agent is over the black line, it senses dark; otherwise, it senses bright. Whenever the agent passes from dark to bright, the sensor sends the micro controller a *bright* interrupt signal. Conversely, when the agent passes from bright to dark, the sensor triggers a *dark* interrupt signal. By default, the agent moves straight unless it receives a *bright* signal, which means that the agent has reached a corner and must start turning right. The second scenario describes this requirement.

The LSC of Fig. 3.3 describes the required behavior whenever the sensor sends a *bright* signal to the micro controller. The specification requires that the agent begin

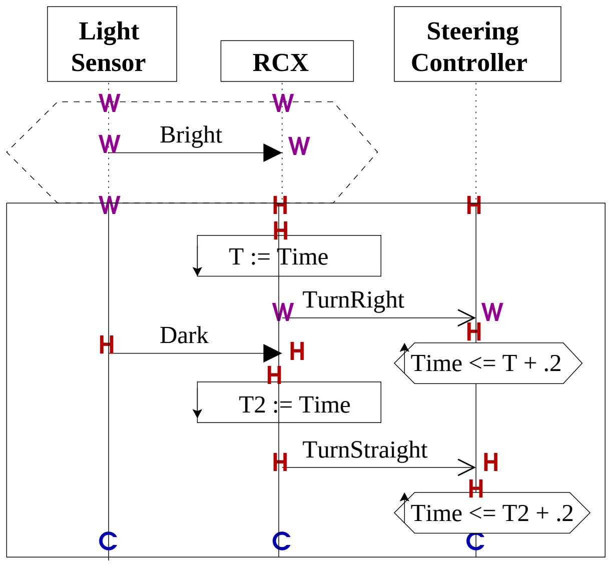


Figure 3.3: Scenario 2: Turning Right

turning no more than 0.2 time units after the sensor sends the signal. Here, we assume that the agent requires zero time to commence a turn and zero time to straighten out the wheels after the turn is complete. If, during the turn, the agent re-intercepts the black line, the light sensor sends a *dark* signal, which prompts the steering controller to end the turn in no greater than 0.2 time units. Notice that this contrived example does not require the *dark* signal to ever occur, as denoted by the cold temperature locations. In this situation, the agent can legally remain in a right turn for the remainder of the run.

Notice that the two scenarios described above are not disjoint with respect to time. That is, they can (and in this case, are *expected* to) execute concurrently: the first LSC turns the motor on and waits ten time units before shutting it off. The second scenario could execute multiple times while the first is executing.

Chapter 4

From Requirements to Specifications

We now describe a process to convert Live Sequence Charts from a visual representation to formal models. The two target models considered here are *deterministic Büchi automata* (DBA) [Büc62] for discrete-time LSCs and *deterministic timed Büchi automata* (DTBA) [AD94] for dense-time LSCs. Both may be used as language acceptors to determine if input event sequences generated by an object system satisfy the LSC requirements.

We proceed by first discussing the single LSC translation to deterministic Büchi automata for discrete-time LSCs, then the single LSC translation to deterministic timed Büchi automata for dense-time LSCs. We then present the multiple-LSC construction which can be applied to either of the two single LSC results.

4.1 Related Work

The earliest known related work is [HK00a], where Harel and Kugler describe how a state-based representation of a single LSC can be combined with others to form a *global state automaton*, formally describing the full set of requirements. The paper describes how the construction would be carried out theoretically, but does not provide full details. It is also in the same work that *satisfiability* of LSC requirements is shown to be equivalent to *consistency*.

In [KW01], Klose and Wittke describe a procedure for converting LSCs to timed automata, where extra features such as *coregions* and *timing annotations* are included. The type of timing constraints used in the above work differ quite fundamentally from those presented in our work. Another distinguishing aspect is that the input alphabet of the automaton in the above work consists of visible events and the internal hidden events. This raises the question of when, how, and by whom the hidden events are expected to originate. According to our approach, these details are taken care of by the construction so that no additional intervention or interpretation is necessary.

In [HM02], Harel and Marelly discuss the use of ordinary LSC assignments and conditions—the same as seen in untimed LSCs—to also express timing constraints. This method of expressing time was then later adopted by a software tool called the Play-Engine [HM03], and is the same approach used here.

In [BH02], Bontemps and Heymans offer a formal semantics based loosely around LSCs. The work covers a set of features that is substantially different from that which is proposed in [DH01], but borrows some main ideas. The authors remove all notions of liveness from conditions, and also disregard the notion of universality or existentiality, instead focusing on charts that are composed by means of parallel and sequential composition. The work also proposes the use *invariants* in addition to the existing chart constructs.

4.2 Single Discrete-Time LSC Specifications

In this section, we first consider a translation from a single LSC to a DBA. Both the untimed and discrete-time constructions are carried out in the same manner, with the only differences being a new assertion $tick$ and a new version of $assign$ and $eval$, all newly equipped to handle time. We later extend the result to consider multiple LSC requirements using the subset of LSC features defined in section 2.2.

4.2.1 State Space

Recall that a simultaneous region (sim-region) is a set of locations which map to some event. All locations in a sim-region satisfy the temporal property $=_T$. The set of sim-regions of a single-LSC automata is characterized by the quotient set of $\texttt{locs}(L)$ by $=_T$, denoted $Sims_L$. Fig. 4.1 shows an example LSC (left) and its corresponding set of sim-regions depicted by solid black rectangles (right).

We write $init(s)$ to assert that a sim-region $s \in Sims_L$ is the *initial sim-region*. That is, the sim-region comprised of the top-most locations of the LSC: $init(s) \Leftrightarrow s = evnts^{-1}(pbegin)$. We define the handy function $g : Sims_L \mapsto E$, mapping every sim-region to an event. Specifically $g(s) = e \Leftrightarrow \forall \ell \in s : evnts(\ell) = e$. Given a sequence

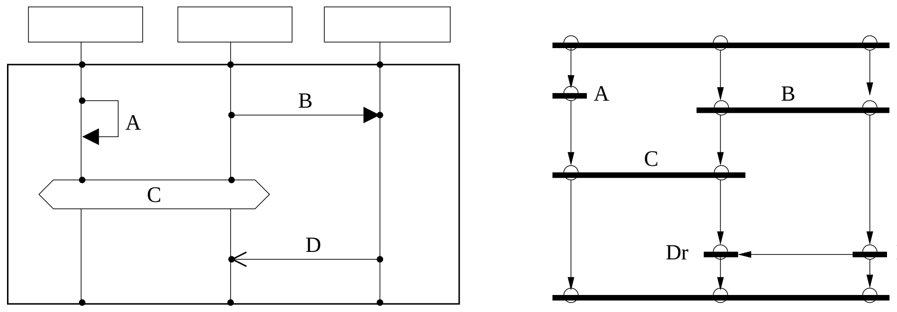


Figure 4.1: LSC Simultaneous regions

of sim-regions $s_k^* = s_0, \ldots, s_k$, we say s_{k+1} is a *successor* of s_k, denoted $s_{k+1} \succ s_k$, iff $simenabled(s_k^*, s_{k+1})$ holds.

4.2.2 Büchi Automata

A Büchi automaton (BA) is a tuple $\mathcal{A} = \langle \Sigma, Q, q_I, \delta, F \rangle$ where:

- Σ is the *alphabet*.
- Q is a finite set of states.
- Q_I is the initial state.
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $F \subseteq Q$ is the set of *final states*.

A *deterministic Büchi automaton* (DBA) is a BA such that $|Q_I| = 1$ and for all states q and actions σ, there exists at most one q' such that $(q, a, q') \in \delta$.

A Büchi automaton decides the inclusion of *infinite words* over Σ by processing individual input symbols sequentially. Execution begins from state $q_i \in Q_I$. Inductively, from a current state q, the next symbol σ is read from the input and if $(q, \sigma, q') \in \delta$ then a transition to q' can be taken. We say that $\mathcal{A}$ is a deterministic Büchi automaton (DBA) iff for all q, σ there exists precisely one q' such that $(q, \sigma, q') \in \delta$. A word is considered *accepting* if every input symbol corresponds to a transition and at least one state in F appears infinitely many times in the word. If $\mathcal{A}$ is some Büchi automaton, then the set of words accepted by $\mathcal{A}$ is the *language of* $\mathcal{A}$, written $L(\mathcal{A})$.

4.2.3 Translation

The set of valuations $\Sigma[V_T]$ for discrete-time variables is infinite because ts-vars can be assigned a system clock value which exceeds any finite bound. This is problematic since Büchi automata require a finite number of states.

Toward a resolution, consider the LSC conditional expression "*Time* $\# \ \varphi \ + x$." Whenever $\# \in \{>, \geq\}$ and the expression is true, the expression remains true for any greater value of *Time*. Likewise, if $\# \in \{<, \leq\}$ and the expression is false, then it remains false for any greater value of *Time*. In general, the expression will always yield the same value for any value of *Time* greater than the largest constant in the requirements.

We take advantage of this property by reformulating the above time expression as an equivalent expression "*Time* $- \ \varphi \ \# \ x$". Instead of representing a ts-var value explicitly, we instead maintain a *timer value* expression *Time* $- \ \varphi$. The timer value is zero when the assignment first occurs. Each occurrence of $Tick$ causes an increase in the timer value. However, the timer value can be bounded by the maximum constant,

$const_{max}$, appearing in any LSC condition over the ts-vars. The assertions $assign_t^*$ and $eval_t^*$ (to be defined shortly) capture these semantics. The assertions work the same as before for the LSC variables and object properties.

We now present one of the contributions of this work:

Theorem 1 *Let $\mathcal{L}_{disc}$ be a single discrete-time LSC and $L(\mathcal{L}_{disc})$ be the language of $\mathcal{L}_{disc}$. Then there exists a DBA $\mathcal{A} = \langle \Sigma, Q, Q_I, \delta, F \rangle$ such that $L(\mathcal{A}) \subseteq L(\mathcal{L}_{disc})$.*

We show this result constructively:

- $\Sigma = E_v$, the visible events of L.
- $Q = \Sigma[V] \times \Sigma[\mathcal{O}] \times 2^{Sims_L} \times \Sigma[V_T]$, the timed cuts of L.

The alphabet is the set of visible events of L. An automaton state consists of an object property valuation component $\Sigma[\mathcal{O}]$, an LSC variable valuation component $\Sigma[V]$, the time-stamp variable component $\Sigma[V_T]$, and the sim-region component 2^{Sims_L}. We let $\pi(q) \in 2^{Sims_L}$ be the set of sim-regions in state q. Intuitively, $\pi(q)$ is the set of sim-regions visited in the past during one scenario execution. A new sim-region is added to the set at each step of the computation.

The initial state of $\mathcal{A}$ is the one where all variables are undefined, all properties are set to an initial value, and the sim-region component is a singleton set referring to the hidden event $pbegin$. Formally,

- $Q_I = \{q \mid init(\Sigma[\mathcal{O}, V, V_T](q)) \;\wedge\; init(\pi(q))\}$

There exists a mapping $f : Q \mapsto$ *TCuts*. We define the set of final states:

- $F = \{q \mid temp(f(q)) \neq H\}$

The acceptance component F enforces the LSC liveness properties discussed in section 2.3. Specifically, that the LSC visits cold and warm cuts infinitely often. In order to express ts-vars as timers, we introduce the following:

$$assign^*_t(a, q) : a.\varphi \in V_T \Rightarrow q(a.\varphi) = 0$$

$$eval^*_t(c, q) : c.\varphi \in V_T \Rightarrow [q(c.\varphi) \neq \bot \;\wedge\; q(c.\varphi)\; c.\#\; c.x]$$

where $\# \in \{=, \neq, <, >, \leq, \geq\}$. The top expression states that the variable φ being assigned in LSC assignment a receives the value 0. The bottom expression evaluates to true if, for a condition c of the form "$\varphi \# x$," variable φ is defined and the expression $\varphi \# x$ is true in state q.

Each transition in δ causes a change of state based on a visible event. When the visible event occurs, a sequence of hidden events is carried out in the order in which the events are permitted by $\succ$. In the event that several possible orderings of events exist, any permissible ordering may be chosen. A heuristic function, for example, may be used for selecting the specific order or set of orderings. More than one ordering may be added, but with the possible side-effect of non-determinism.

Given a state $q \in Q$ and visible event $e \in E_v$, the *successor sequence* of (q, e) is a sequence of sim-regions $s^*_k = s_1, \ldots, s_k$ satisfying:

1. $s_1 \in \pi(q)$

2. $\forall i\ (1 < 1 \leq k) : s_i \succ s_{i-1}\ \wedge\ g(s_i) \in E_h$

Momentarily, we discuss how to encapsulate the semantics of these hidden events into a single automaton transition.

A successor sequence $s_1, \ldots, s_k$ is *maximal* if for all sim-regions s_{k+1} such that $s_k \succ s_{k+1}$, it is the case that $g(s_{k+1}) \in E_v$. This means that only visible events are reachable from s_k. Given a successor sequence for (q_1, e) we arrive at a state q_k by inducing a sequence $q_1, \ldots, q_k$ where for all $i < k : q_{i+1} = \mathit{TNext}(q_i, g(s_i))$. We call (q_1, e, q_k) a *successor sequence transition*. A successor sequence transition whose successor sequence is maximal is called a *maximal successor sequence transition*.

- $\delta = \delta^F \cup \delta^B$ is defined below.

δ^F is the set of maximal successor sequence transitions (q_1, e, q_k), each with precisely one maximal successor sequence $s_1, \ldots, s_k$ of (q_1, e) inducing a corresponding state sequence $q_1, \ldots, q_k$, such that the following requirements are satisfied for all $1 < i \leq k$:

1. $\mathit{simenabled}(\pi(q_{i-1}), s_i)$

   Semantics for enablement (cf. secs. 2.7.4 and 3.3.4, requirement 1).

2. $\forall a \in A : [\texttt{locs}(a) \subseteq s_i\ \wedge\ a.\varphi \in V] \Rightarrow [\mathit{assign}(f(q_{i-1}), f(q_i), a.\varphi, a.x)]$

   Untimed assignments (cf. sec. 2.7.4, requirement 2).

3. $\forall a \in A : [\texttt{locs}(a) \subseteq s_i] \Rightarrow \mathit{assign}_t^*(a, f(q_i))$

   Discrete-time assignments (cf. sec. 3.3.4, requirement 2).

4. $\forall \varphi \in (V \cup \mathit{Props}) : [[\forall a \in A : a.\varphi \neq \varphi \vee \texttt{locs}(a) \nsubseteq s_i] \Rightarrow$ $\mathit{unch}(f(q_{i-1}), f(q_i), \varphi)]$

   Unchanged untimed variable valuations (cf. sec. 2.7.4, requirement 3).

5. $\forall \varphi \in V_T : [[\forall a \in A : a.\varphi \neq \varphi \vee \texttt{locs}(a) \nsubseteq s_i] \Rightarrow \mathit{unch}_t(f(q_{i-1}), f(q_i), \varphi)]$

   Unchanged timed variable valuations (cf. sec. 3.3.4, requirement 3).

6. $\forall c \in C : [\texttt{locs}(c) \subseteq s_{i-1}] \Rightarrow [\mathit{eval}(f(q_{i-1}), c) = \text{TRUE}]$

   Untimed (hot and cold) conditions (cf. sec. 2.7.4, requirement 4).

7. $\forall c \in C : [\texttt{locs}(c) \subseteq s_{i-1}] \Rightarrow [\mathit{eval}_t^*(f(q_{i-1}), c) = \text{TRUE}]$

   Timed (hot and cold) conditions (cf. sec. 3.3.4, requirement 4).

δ^B is the set of maximal successor sequence transitions (q_1, e, q_{init}) each with precisely one maximal successor sequence $s_1, \ldots, s_k$ of (q_1, e) inducing a corresponding state sequence $q_1, \ldots, q_k$, such that *at least one* of the following requirements are satisfied for all $1 < i \leq k$:

1. $\exists c \in C : [\texttt{locs}(c) \subseteq s_{i-1} \wedge \mathit{temp}(c) = C \wedge \mathit{eval}_t^*(f(q_i), c) = \text{FALSE}]$

   False timed cold conditions (cf. sec. 3.3.4, requirement 5).

2. $\exists c \in C : [\texttt{locs}(c) \subseteq s_{i-1} \wedge \mathit{temp}(c) = C \wedge \mathit{eval}(f(q_{i-1}), c) = \text{FALSE}]$

   False untimed cold conditions (cf. sec. 2.7.4, requirement 6).

3. $\mathit{temp}(f(q_i)) = C$

   Reaching a cold cut (cf. sec. 2.7.4, requirement 5).

and *all* of these requirements hold:

4. $\forall c \in C : [\mathtt{locs}(c) \subseteq s_{i-1} \ \wedge \ \mathit{temp}(c) = H \Rightarrow \mathit{eval}(f(q_{i-1}), c) = \text{TRUE}]$

   Hot untimed conditions must be true (cf. sec. 2.7.4, requirement 4).

5. $\forall c \in C : [\mathtt{locs}(c) \subseteq s_{i-1} \ \wedge \ \mathit{temp}(c) = H \Rightarrow \mathit{eval}_t^*(f(q_{i-1}), c) = \text{TRUE}]$

   Hot timed conditions must be true (cf. sec. 3.3.4, requirement 4).

6. $\forall v \in V : f(q_{init})(v) = \perp$

   Uninitializing untimed variables (cf. sec. 2.7.4, requirement 7).

7. $\forall v \in V_T : f(q_{init})(v) = \perp$

   Uninitializing ts-vars (cf. sec. 3.3.4, requirement 6).

Turning to correctness, we show that the initiality and consecution requirements of section 3.3.5 are met and show that the automaton captures discrete-time LSC computations.

The initiality requirement is satisfied by the singleton Q_I, whereas consecution is satisfied by the forward and back edge sets δ^F and δ^B, respectively. Forward edges are characterized by assignments and true conditional evaluations. Back edges are characterized by entry into cold cuts and false cold conditional evaluations. We list the circumstances under which edges in δ^F and δ^B are constructed above; the reader may easily confirm the correspondence to the requirements in sections 2.7.4 and 3.3.4.

LSC computations are captured by the edges above in combination with the liveness properties which are expressed by means of state temperature. Specifically, infinitely many visits to warm and cold states are considered satisfying behavior, and the set of final states F capture this requirement.

$\square$

For the purpose of maintaining determinism, the construction requires precisely one maximal successor sequence. It is not necessarily the case that more than one sequence will result in non-determinism. Indeed, there may exist multiple permissible sequences, each carrying out the events in different orders, whose final state q_k is the same regardless of the order. However, the redundant sequences are unnecessary since all ultimately lead to q_k. On the other hand, non-determinism is possible if the multiple orderings induce distinct terminal states. A definitive ordering must therefore be chosen, using heuristics for example.

Our approach deviates from the semantics of smart play-out in two general ways. First, smart play-out does not necessarily execute hidden events at the same time instant. This is because waiting (or not waiting) to execute an assignment or condition may change a future outcome (e.g., whether a violation occurs later). The second point of differentiation is that paths exist for all permissible orderings in smart play-out, so that the model-checker can discover any ordering which won't cause a violation.

4.2.4 Example

Fig. 4.2 shows the general automaton structure induced by the translation procedure, given the previous example of Fig. 4.1 as input. The top-most state $\{I\}$ reflects the initial hidden event $pbegin$. This state and the bottom-most state refer to the same state, thereby introducing a cycle. To reduce clutter, this example illustrates only the forward edges.

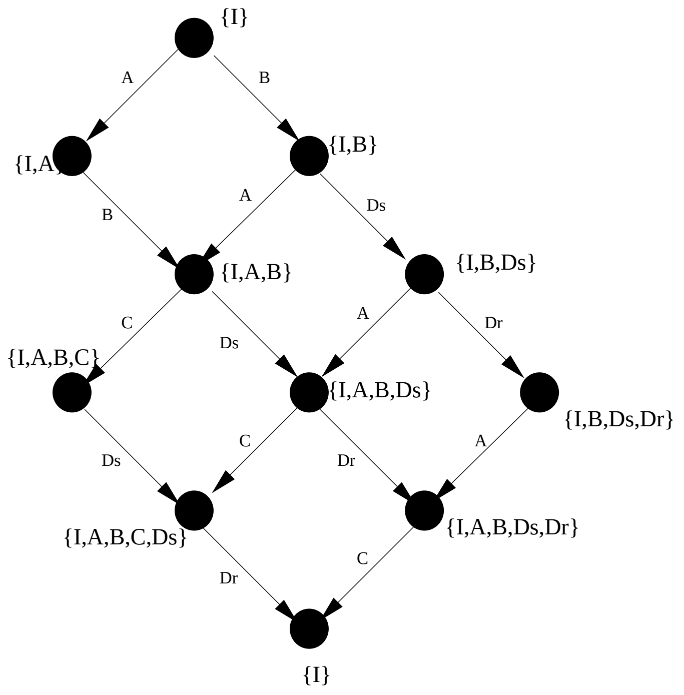


Figure 4.2: Büchi Automaton of Fig. 4.1

4.3 Dense-Time LSC Specifications

The discrete-time approach was possible due to the use of timers in establishing an upper bound on the infinite discrete set of ts-var valuations. The same approach under dense time doesn't work since there are still infinitely many valuations, despite the upper bound. In this section, we show how the language of dense-time LSC requirements can be translated into a finite-state specification.

For the purposes of this section, we consider *deterministic timed Büchi automata* (DTBA) specifications, first introduced in the seminal paper of [AD94]. The paper more generally introduces dense time to a collection of previously existing formal models which recognize ω-regular languages. Specifically, *timed Büchi automata* (TBA) and *timed Muller automata* (TMA) are considered and shown to be equivalent in expressive power under non-determinism. Deterministic models, such as deterministic timed Büchi automata (DTBA) and deterministic timed Muller automata (DTMA), are also considered. It is shown that DTBA—the target of our upcoming LSC translation—recognize languages which are strictly less expressive than those recognized by DTMA and are closed under union and intersection, but not complement. However, a straightforward translation from DTBA to DTMA is possible, which admits closure under complement.

It turns out that timed automata are particularly useful for verification and synthesis. This is due to the *region construction* proposed in [AD94], which expresses the infinite-state timed automata as a finite state untimed model which is equivalent with respect to the verification and synthesis problems. Timed automata are often used for *model-checking*, in which one seeks to discover whether various *properties* hold for all or some behaviors expressed by the language of a timed automata. Therefore, if we can only express LSCs in terms of timed automata, the ability to perform model checking on timed automata comes for free for those model checkers supporting timed automata. One such model checker is Uppaal [LPY95].

4.3.1 Timed Automata

Before proceeding with the LSC to timed automata translation, we first give a brief overview of the timed automata model introduced in [AD94]. *Deterministic timed Büchi automata* (DTBA) are Büchi automata supplemented with a finite set of *clocks* whose values range over the non-negative reals. Each clock has a *valuation* which begins initially at zero and increases continuously at a rate of 1. A *state* of the automaton is a discrete state (from a finite set of discrete states) and a valuation mapping every clock to a non-negative real time value.

Each transition consists of a *guard* (conditional expression over the clock valuations) and a set of clocks whose valuations are reset to zero if the guard is true and the transition is taken. A transition may be taken from an origin state whenever the guard evaluates to true in the state. The transition leads into some destination state state whose clock valuations reflect the clock resets from the transition, but whose other clock valuations are identical to the origin state.

The *system clock* is a special clock which is never reset and therefore always reflects absolute time elapsed since the beginning of the run. A run is a sequence of *timed events* which cause some *accepting state* to be reached infinitely often. A timed event is an event (action) associated with some system clock valuation, meant to refer to the absolute time at which the event occurred. Upon the occurrence of each timed event, all clock valuations are updated to reflect the elapsed time since the last timed event.

Formally, a timed Büchi automata (TBA) is a tuple $\mathcal{A} = \langle \Sigma, Q, Q_I, Cl, E, F \rangle$ consisting of the following components:

- Σ — a finite set of *actions*.
- Q — a finite set of *states*.
- $Q_I \subseteq Q$ — a set of *initial states*.
- Cl — a finite set of *clocks* each ranging over $\mathbb{R}_{\geq 0}$.

- $\delta \subseteq Q \times \Phi(Cl) \times \Sigma \times 2^C \times Q$— a set of *transitions*, where $\Phi(Cl)$ is a set of *guards* (conditional expressions) over Cl.
- F — a set of *final* states.

An edge $(q, g, a, r, q') \in \delta$ leads from state q into q' by the input action a, guarded by $g \in \Phi(Cl)$, and *resets* the clocks in $r \subseteq Cl$. A guard is defined inductively by $\delta = x \# c \mid c \# x \mid \neg\delta \mid \delta \wedge \delta$, where $x \in Cl$, $c \in \mathbb{Q}$, and $\# \in \{<, \leq, \geq, >\}$. Note that the constants are rational, which is required for the aforementioned region construction to work.

A *deterministic timed Büchi automaton* (DTBA) is a TBA in which $|Q_I| = 1$ and for all states s, actions a and edge pairs $\langle q, -, a, -, g_1\rangle$ and $\langle q, -, a, -, g_2\rangle$, guard $g_1 \wedge g_2$ is unsatisfiable.

A *timed word* is a pair $(\sigma, \tau) = (a_1, t_1), (a_2, t_2), \ldots$ where σ is an infinite sequence of actions and τ is an infinite sequence of non-negative real numbers satisfying strict monotonicity ($\tau_i < \tau_{i+1}$ for all $i \geq 1$) and progress (for all non-negative reals t, there exists an i such that $\tau_i > t$).

Given a *timed word* $w = (\sigma, \tau) = (a_1, t_1), (a_2, t_2), \ldots$, a *run* of $\mathcal{A}$ over w is an infinite sequence $r = (q_0, \nu_0), (q_1, \nu_1), \ldots$ where each q_i is a state, and each ν_i is a real valuation of the clocks in Cl. We say that a valuation *satisfies* a guard, written $\nu \models g$ if the guard is true under valuation ν. We write $\nu + t$ to denote the clock valuation ν with t added to each value. We require:

1. Initiality: $q_0 \in Q_I$ and for all $x \in Cl$, $\nu_0(x) = 0$.
2. Consecution: $\forall i > 0$ we have $(q_{i-1}, q'_i, a_i, g_i, r_i) \in \delta$ s.t.:

- $(\nu_{i-1} + t_i - t_{i-1}) \models g_i$.
- For all $x \notin r$, $\nu_i(x) = \nu_{i-1}(x) + t_i - t_{i-1}$.
- For all $x \in r$, $\nu_i(x) = 0$.

4.3.2 Translation

The translation to timed automata follows similarly to the discrete-time case, except that assignments and conditions over ts-vars are not expressed through discrete variables by means of assertions $eval_t^*$, $assign_t^*$, and $unch_t$ but rather through *guards* and *clock resets*. This introduces interesting subtle complexities which are not present in the discrete-time construction.

Undefined Time-stamp Variables

We have seen in section 3.4.1 that ts-vars take on *undefined* values ($\perp$) at certain points during execution. The significance is that any undefined ts-var appearing in an LSC condition results in a false evaluation, by definition. As we will see, ts-vars are implemented in our construction by means of timed automata guards. However, timed automata have no notion of undefined clock values and therefore no way to recognize undefined clocks inside a guard, since clocks always map to *some* non-negative real value.

We can express undefined clock values in timed automata by introducing an array of Boolean flags as part of the discrete state space—one flag for each clock. A true flag valuation indicates that a particular clock value is "defined" and that any transition leaving such a state whose guard contains a reference to that clock should be evaluated normally. A false flag valuation, on the other hand, represents an “undefined” value. For each ts-var, it is straightforward to set the corresponding flag to true whenever the ts-var is first assigned in an LSC and false whenever a back-edge to the initial state is taken.

Supposing the aforementioned array is present, let us discuss how the information would be used: we first construct the automaton from the LSC under the assumption that all ts-vars are defined, ignoring the flags. After the automaton is built, we remove transitions in the following way. Given a discrete state in which some clock x is flagged as undefined (false), we *remove* all transitions leaving that state in which x appears in the guard. The rationale is that the LSC condition—and hence, the corresponding automaton guard—would always evaluate to false, making it impossible to traverse the edge.

Below, we will present a translation from dense-time LSCs to a timed automaton. Rather than complicating the construction with the above details, we instead propose to carry out the edge removal as a post-processing step. It is easy to see that the resulting automaton will preserve the semantics of undefined ts-vars, since only guards which test the values of “defined” ts-vars will remain in the final automaton.

Static Evaluation

One limitation of timed automata transitions is that clocks may be reset only *after* the guard has evaluated to true. This presents a complication from the standpoint of our construction, since the successor sequence may dictate an order which the automaton cannot handle—specifically, an assignment followed by a condition over the same ts-var. In this case, we can *statically evaluate* the condition and modify the transitions to reflect the evaluation. This is possible since ts-var conditional expressions of the form "*Time* $\#$ $\varphi + const$," can be reduced to the expression *Time* $\#$ *Time* $+ \, const$.

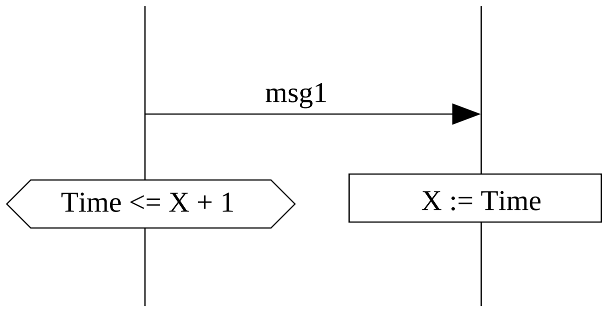


Figure 4.3: Timed Automata Successor Sequence Problem

To solidify this point, consider the simple example of Fig. 4.3. At the moment $msg1$ occurs, an assignment and condition execute in some order. Two successor sequences are possible: the assignment followed by the condition, and vice versa. In the case that the condition precedes the assignment, no special handling is necessary since the automaton edge will already perform the actions in the desired order. On the other hand, executing the assignment before the condition requires static evaluation: by substituting *Time* for X, we arrive at *Time* $\leq$ *Time* $+ 1$, which is true. Consequently, the assignment will be carried out by the automaton by means of a clock reset, whereas the conditional expression will be removed.

At first, it may seem reasonable to proceed in a manner similar to undefined ts-vars: construct the rest of the automaton and then remove any edges in which false statically evaluations exist. Although possible to carry out this way, we avoid this approach for two reasons: first, it is possible to accomplish this without extending the state space with more flag variables. Secondly, we feel it is more intuitive to the reader if this is handled from within the construction itself, rather than as a post-processing step.

Toward solving this issue, we introduce a predicate $staticeval : C \mapsto \mathbb{B}$, which given any condition, will evaluate the condition under the assumption that *Time* $= \varphi$. Whenever it is necessary to statically evaluate a timed condition, the outcome of this predicate will be "hard-coded" into the timed automaton guard. This can be achieved by adding conjuncts which make the guard unsatisfiable for false outcomes (e.g., $c < x \ \wedge \ c > x$), or by not adding conjuncts for true outcomes. Equivalently, the transition can be removed.

Given a successor sequence $s_k^* = s_1, \ldots, s_k$ for (q_1, e), it is essential that the automaton transitions respect the ordering in which assignments and conditions are carried out according to s_k^*. It is possible to determine the outcome of any condition over the discrete variables at the time of construction. This is accomplished by starting at state q_1, applying the assignment or condition semantics from sim-region s_2 to arrive at q_2, and so on.

It turns out for the following construction that we need to know the first position among the sim-regions in s_k^* at which a hot condition over the discrete variables evaluates to false, if any. Similarly, we need to know the first position in s_k^* at which any cold condition over the discrete variables evaluates to false, if any. The following reasoning applies: if the first hot condition over the discrete variables which evaluates to false occurs at position j, then a transition to the initial state can still be taken if there exists a false cold condition over the ts-vars at any position $i < j$ such that the ts-var condition evaluates to false. Intuitively, if a false cold condition occurs before the false hot condition, the chart may close without violation—this is a legal transition to the initial state.

Conversely, if we know that a cold condition over the discrete variables evaluates to false at position j in the successor sequence, then the transition can be taken, provided that all hot conditions over the ts-vars at positions $i < j$ evaluate to true. Intuitively, if a hot condition evaluates to false prior to j then this is a violation of the requirements—no transition to the initial state exists in this case.

We now present another contribution of this work, in which we offer a translation from single dense-time LSCs to a deterministic timed Büchi automaton:

Theorem 2 *Let $\mathcal{L}_{dens}$ be a single dense-time LSC and $L(\mathcal{L}_{dens})$ be the language of $\mathcal{L}_{dens}$. Then there exists a DTBA $\mathcal{A} = \langle \Sigma, Q, Q_I, Cl, \delta, F\rangle$ such that $L(\mathcal{A}) \subseteq L(\mathcal{L}_{dens})$.*

By way of construction, we define $\mathcal{A}$ as follows:

- $\Sigma = E_v$
- $Q = \Sigma[\mathcal{O}] \times \Sigma[V] \times 2^S$
- $Q_0 = \{q \mid init(\Sigma[\mathcal{O}, V, V_T](q)) \ \wedge \ init(\pi(q))\}$
- $Cl = V_T$
- $F = \{q \mid temp(f(q)) \neq H\}$
- δ is the set of transitions $\delta^F \cup \delta^B$ defined below.

δ^F is the set of transitions (q_1, g, e, r, q_k), each with precisely one maximal successor sequence transition (q_1, e, q_k) and each having a maximal successor sequence $s_k^* = s_1, \ldots, s_k$ of (q_1, e) such that for all i $(1 < i \leq k)$:

1. $\mathit{simenabled}(\pi(q_{i-1}), s_i)$

   Semantics for enablement (cf. secs. 2.7.4 and 3.3.4, requirement 1).

2. $\forall a \in A : [\mathtt{locs}(a) \subseteq s_i \ \wedge \ a.\varphi \in V] \Rightarrow [\mathit{assign}(f(q_{i-1}), f(q_i), a.\varphi, a.x)]$

   Untimed assignments (cf. sec. 2.7.4, requirement 2).

3. $\forall \varphi \in (V \ \cup \ \mathit{Props}) : [[\forall a \in A : a.\varphi \neq \varphi \ \vee \ \mathtt{locs}(a) \nsubseteq s_i] \Rightarrow$ $\mathit{unch}(f(q_{i-1}), f(q_i), \varphi)]$

   Unchanged untimed variable valuations (cf. sec. 2.7.4, requirement 3).

4. $\forall c \in C : [\mathtt{locs}(c) \subseteq s_{i-1}] \Rightarrow [\mathit{eval}(f(q_{i-1}), c) = \text{TRUE}]$

   Untimed (hot and cold) conditions (cf. sec. 2.7.4, requirement 4).

5. $r = \bigcup_{i=1}^{k} \ \{a.\varphi \mid a \in A \ \wedge \ \mathtt{locs}(a) \subseteq s_i \ \wedge \ a.\varphi \in V_T\}$

6. $g = \bigwedge_{i=1}^{k} \ \{\mathit{guard}(c) \mid c \in C \ \wedge \ \mathtt{locs}(c) \subseteq s_i\}$

where:

$$\mathit{guard}(c) = \begin{cases} c.\varphi \ \# \ c.x & \text{if } c.\varphi \text{ is not in an assignment before } c \text{ in } s_k^* \\ \mathit{staticeval}(c) & \text{if } c.\varphi \text{ is in an assignment before } c \text{ in } s_k^* \end{cases}$$

δ^B is the set of back-edges. Given the maximal successor sequence transition (q_1, e, q_k) and one maximal successor sequence $s_k^* = s_1, \ldots, s_k$ of (q_1, e), let:

- $x_1, \ldots, x_p$ be the list of all *cold* ts-var condition positions in s_k^*.

- $y_1, \ldots, y_q$ be the list of all *hot* ts-var condition positions in s_k^*.

- $\overline{H}$ ($\overline{C}$ resp.) be the position of the first false hot (cold resp.) condition in s^*_k over the LSC variables and object properties, or 0 if none exists.

Let $(q_1, g, e, r, q_{init}) \in \delta^B$ if $\overline{C} \neq 0$ where for some i $(i \leq k)$ and all j $(j < i)$, all of the following requirements hold:

1. $[\exists c \in C : \texttt{locs}(c) \subseteq s_{i-1} \ \wedge \ \mathit{temp}(c) = C \ \wedge \ \mathit{eval}(f(q_{i-1}), c) = \text{FALSE}] \ \vee$ $\mathit{temp}(f(q_i)) = C$

   Cold cuts and false untimed cold conditions (cf. sec. 2.7.4, requirements 5 & 6).

2. $\forall c \in C : [\texttt{locs}(c) \subseteq s_{j-1} \ \wedge \ \mathit{temp}(c) = H \Rightarrow \mathit{eval}(f(q_{j-1}), c) = \text{TRUE}]$

   Hot untimed conditions must be true (cf. sec. 2.7.4, requirement 4).

3. $\forall v \in V : f(q_{init})(v) = \perp$

   Uninitializing untimed variables (cf. sec. 2.7.4, requirement 7).

4. $\forall v \in V_T : f(q_{init})(v) = \perp$

   Uninitializing ts-vars (cf. sec. 3.3.4, requirement 6).

5. $r = \emptyset$

6. $g = \bigwedge_{j=1}^{q} \ \{\mathit{guard}(c(s_{y_j})) \mid \overline{C} = h \neq 0 \ \wedge \ y_j < h\}$

Let $\{(q_1, g_1, e, r, q_{init})_1, \ldots, (q_1, g_p, e, r, q_{init})_p\} \subseteq \delta^B$ s.t. for all $1 < i \leq p$:

1. Requirements 2, 3 and 4 above hold.

2. $\overline{C} = 0 \wedge (\overline{H} = 0 \vee x_i < \overline{H})$

3. $r_i = \emptyset$

4. $g_i = \neg guard(c(s_{x_i})) \ \wedge \ \bigwedge_{j=1}^{x_i-1} guard(c(s_{x_j}))$

where $c(i)$ refers to the condition at position s_i in the successor sequence.

We now show that the transitions of the timed automaton faithfully simulate a dense-time LSC computation. In the above construction, we describe the circumstances under which automaton edges are created in three cases: one for forward-edges and two for back-edges. We show that these cases collectively meet the requirements for dense-time LSCs as defined in section 3.4.1. There are five requirements.

Requirement 1 states that all untimed requirements from section 2.8.4 hold, and requirements 2-5 are specific to dense-time. The general construction technique is substantially the same as in Theorem 1 for the untimed requirements, with the only difference being a case split on the set of back-edges. It is straightforward to confirm that all untimed requirements hold among the three cases.

For each ts-var assignment, rather than storing the fixed value of the system clock at the time of the assignment, we instead reset a clock and refer to the elapsed time inside the subsequent conditions. LSC assignments are carried out by resetting clocks and conditions are implemented as guards. The reader can easily verify that the assertions $assign_t^*$ and $eval_t^*$ capture the formal semantics, thereby satisfying requirements 2-4. Requirement 5 states that all ts-vars are undefined in the untimed initial state, and can be met by applying the post-processing procedure described earlier.

As in Theorem 1, forward-edges capture successor sequences which terminate in a state from which only visible events are permissible. Unlike Theorem 1, however, there are two sets of back-edges which lead to the untimed initial state. The outcome of a condition over a discrete variable is known at the time of the automaton construction, whereas the outcome of a condition over the ts-vars is not known until the automaton is deciding membership of a timed input word. If a condition over a discrete variable evaluates to false within the successor sequence, there are two cases depending on whether the condition is hot or cold.

The first case identifies the first cold condition over a discrete variable in the successor sequence which evaluates to false. An automaton transition leading to the untimed initial state is created and the corresponding guard checks that all hot conditions over ts-vars leading up to this point are satisfied. The second case pertains to the first hot condition over a discrete variable in the successor sequence which evaluates to false. If there exists some false cold condition over the ts-vars which occurs before that point in the successor sequence, however, then a transition to the untimed initial state is still possible, thereby avoiding violation. Multiple transitions to the untimed initial state are therefore constructed—one for every cold condition over the ts-vars leading up to that point. The corresponding guards test for the falsehood of each cold condition over the ts-vars.

By definition, all timed automata runs satisfy initiality, consecution, monotonicity and progress, and are therefore also dense-time LSC runs. To show that they are also dense-time LSC computations, we observe that the set F of final states above captures the requirement of infinitely many visits to a cold or warm state.

$\square$

4.4 Multiple LSC Specifications

We turn to the construction of a generalized deterministic timed Büchi automaton which accepts the language of multiple timed LSCs $\langle L_1, \ldots, L_m \rangle$. The discrete case is not treated here because it adds extra detail without offering any new insight. A straightforward automata product construction cannot be immediately applied, for the reasons explained below.

First, one may think of the multiple-LSC construction as a synchronous composition of visible events, but an asynchronous composition of successor sequences. That is to say, if multiple LSCs are enabled to carry out the same visible event, then an occurrence of that event will cause all enabled input automata to transition simultaneously. The successor sequence which then follows amounts to an asynchronous interleaving of the hidden events among the input automata. To maintain a deterministic construction, a single interleaving for each (q, e) pair must be fixed. The duality between the asynchronous and synchronous semantics requires special handling, which we address in the next section.

Next, the object property valuations should be shared between the LSCs in the sense that a change in an object's property valuation in one scenario should be reflected immediately in all others. A brute-force product construction would wrongly produce independent object valuations. This can be addressed rather easily by redefining the state space so that all object valuations are shared, but everything else—the LSC variables, and ts-vars—are owned by each individual LSC.

The third reason is that *memory* is required for keeping track of whether each LSC passes through a final state infinitely often. Rather than add memory variables explicitly, we instead employ a *generalized Büchi* acceptance condition in conjunction with a timed transition system, which yields a generalized timed Büchi automaton. This is an automaton whose acceptance component F is instead expressed as a set of accepting sets $F_1, \ldots, F_n$. The requirement is that at least one state in *each* of the sets occur infinitely often during a run. Generalized timed Büchi automata can be expressed as an ordinary timed Büchi automata using known techniques.

4.4.1 Translation

We present the main translation result and additional contribution of this work:

Theorem 3 *Let $\mathcal{L}_{multi} = \langle L_1, \ldots, L_n \rangle$ be dense-time LSC requirements and $L(\mathcal{L}_{multi})$ be the language of $\mathcal{L}_{multi}$. Then there exists a generalized DTBA $\mathcal{A} = \langle \Sigma, Q, Q_I, \delta, F^n \rangle$ such that $L(\mathcal{A}) \subseteq L(\mathcal{L}_{multi})$.*

A single LSC is defined by $\Gamma_j = L_j.\Sigma[V] \times L_j.\prod_{i \in I} \texttt{locs}(i) \times L_j.\Sigma[V_T]$. This comprises the set of variable valuations, instance-to-location mappings and ts-var valuations for LSC j. We denote by $\Gamma_i(q)$ the current valuation of LSC i in state q, and by $\Gamma(q)$, the current valuation of all LSCs in state q. We construct $\mathcal{A}$ as follows:

- $\Sigma = \bigcup_{i=1}^{n} L_i.E_v$
- $Q = \Sigma[\mathcal{O}] \times \Gamma_1 \times \cdots \times \Gamma_n$

  We define a partial function $f : Q \times \{1, \ldots, n\} \mapsto$ *TCuts* which given a state and LSC number, yields a timed cut.

- $Q_I = \{q \in Q \mid \forall i : init(f(q,i))\}$
- $F^n = (F_1, \ldots, F_n)$ where $F_i = temp(f(q,i)) \neq H$
- δ is the set of transitions $\delta^F \cup \delta^B$.

We start by defining the notion of successor sequences for multiple-LSC requirements. Specifically, we build up to the definition of *maximal composed successor sequence transition* in a manner similar to the non-composed version. The contents of sets δ^F and δ^B then follow from the construction of Theorem 2.

The notion of successor sequences is similar to the single-LSC case, except that there exist n enablement assertions, distinguished by $simenabled_i$ for $1 \leq i \leq n$. We can express the notion of enablement with respect to a sequence of sim-regions belonging to LSC i using the notation $\succ_i$. In a multiple-LSC setting, successor sequences are an interleaving between the permissible orderings according to each $\succ_i$. The sequence is thus a hybrid of sim-regions among the LSCs.

As in the single-LSC case, the chosen interleaving of assignments and conditions over the variables and object properties are respected by the maximal composed successor sequence. There can exist greater than one maximal composed successor sequence. In order to preserve determinism, precisely one must be selected since any larger subset of interleavings may result in a non-deterministic construction. As before, a heuristic function may be employed to choose among the available interleavings.

Let $i_1, \ldots, i_k$ be a sequence of numbers ranging over $\{1, \ldots, n\}$, where each refers to the number of an LSC in the requirements. We write s^{i_j} to refer to a sim-region of LSC i_j. Given a state $q \in Q$ and visible event $e \in E_v$, the *composed successor sequence* of (q, e) is a sequence of sim-regions $s^* = s_1^{i_1}, \ldots, s_k^{i_k}$ satisfying:

1. $\exists s_m^i \in s^* : s_m^i \in \pi(f(q, i))$
2. $\forall s_m^i \in s^* : s_m^i \in \pi(f(q, i)) \ \wedge \ s_m^i \succ_i s_n^i \Rightarrow g(s_n^i) = e$
3. $\forall s_m^i \in s^* : s_m^i \in \pi(f(q, i)) \ \wedge \ s_m^i \not\succ_i s_n^i \Rightarrow g(s_n^i) \in E_h$
4. $\forall m, n \nexists p : m < p < n \ \wedge \ s_m^i, s_p^i, s_n^i \in s^* \Rightarrow s_m^i \succ_i s_n^i$

The first statement says that each LSC has a sim-region in state q. The second statement says that all sim-regions which are immediate successors of sim-regions in s^* are associated with visible event e. Each of these events will ultimately cause progress to be made in any LSC in which e is enabled. The third requirement states that any sim-region other than immediate successors of s will refer to hidden events. This prevents events other than e (e.g., other messages) from occurring during the sequence. The fourth statement requires the sequence to satisfy $\succ_i$ for all LSCs $1 \leq i \leq n$.

A composed successor sequence $s^* = s_1^{i_1}, \ldots, s_k^{i_k}$ is *maximal* if for all i and for all x, we have $s_{\max(j)}^i \succ_i s_x^i \ \wedge \ g(s_x^i) \in E_v$. This means that all reachable hidden events have been exhausted and only visible events are successors.

Given a composed successor sequence for (q_1, e) we arrive at a state q_k by inducing a sequence $q_1, \ldots, q_k$ where $\forall i < k \ \exists t_i : q_{i+1} \in \mathit{TMNext}(q_i, (g(s_i), t_i))$. We call (q, e, q_k) a *composed successor sequence transition.* As before, a successor sequence transition whose successor sequence is maximal is called a *maximal composed successor sequence transition.*

Now that we have formally defined the notion of a maximal successor sequence transition, we construct δ^F and δ^B according to Theorem 2. For each (q, e), limiting the number of maximal composed successor sequences to one essentially reduces the size of the language by discarding other paths. This reduces the correctness proof of this construction to that given in Theorem 2.

□

4.5 Non-Deterministic Hidden Events

All of the constructions offered in this chapter assume that assignments and conditions occur at deterministic points in time. Specifically, at the same time instant that they become enabled, which is also the same time instant at which the preceding visible event occurs. The overall semantics are generally the same, with the differences resting mainly in the minor details.

We've also noted that smart play-out differs from our approach in that it uses a less restrictive semantics for determining the time at which to execute assignments and conditions. It is reasonable to ask why we have not chosen similar semantics for our constructions in this chapter.

Intuitively, such a construction would require a means to execute hidden events temporally *between* visible events, rather than at the same time instant. A so-called *silent transition* (otherwise known as ϵ-transition) variant of timed automata would seem necessary so that transitions could fire without an accompanying event. As one might expect, such an addition introduces non-determinism. It turns out that there are other side-effects in the case of dense-time.

The original timed automata work of [AD94] does not include any notion of silent transitions. The issue of adding silent transitions is considered in [BGP96], and concludes that doing so can add expressiveness to the language. That is, it can result in an automaton whose language cannot be expressed by any timed automaton without silent transitions. Later, in [DGP97], it is further shown that the introduction of silent transitions does not add expressiveness if no clocks are reset on the transition. Moreover, silent transitions with resets may also occur, but may not appear on any directed cycle.

It seems problematic to implement the semantics of non-deterministic LSC assignments in timed automata. Specifically, since a dense-time assignment is implemented as a clock reset, a silent transition with a clock reset is required. All paths in an LSC automata—according to our subset of features—lie on a directed cycle, thereby making the above results unusable for relating the expressiveness of the automaton with silent transitions to one without.

We believe it may be still possible to prove LSC automata, in particular, exempt from the problems above despite the presence of cycles. Nevertheless, additional problems exist. Recall that a successor sequence can, in the general case, contain an arbitrary interleaving of assignments and conditions. Although, according to the non-deterministic interpretation, it is unnecessary to group together the successor sequence temporally, a construction method for dealing with arbitrary sequences is still necessary. However, it is unknown to us at present whether an algorithm for removing silent transitions from arbitrary successor sequences is decidable.

Chapter 5

Automata-Based Synthesis

Going directly from requirements to a correct implementation has long been a "holy grail" for system and software development. According to this vision, instead of implementing a system and then working hard to apply testing and verification methods to prove system correctness, a system is rather built correctly by construction. Synthesis is particularly challenging for reactive systems, in which the synthesized system must satisfy the requirements for any possible behavior of an external environment [ALW89, Dil89, PR89a].

The problem of constructing reactive systems directly from formal requirements dates back to Church's problem in 1963. Various solutions to the problem have since been proposed; for example, see [BL69, Rab72]. Due to the widespread interest in Linear Temporal Logic (LTL), synthesis using LTL formulas as requirements has also been considered [PR89a]. In the general case, time complexity for most of these solutions is intractable due to the theoretical double exponential lower bound [PR89a]. Despite this, recent work focuses on various subsets of LTL [AMPS98, AT04] whose time complexity is suitable for building tools of practical use.

It turns out that one particular subset is sufficient for the purposes of synthesizing from many interesting LSC requirements. The subset in question is the recurrence property seen previously—“infinitely often, the LSC must be in its prechart.” Moreover, the synthesis problem for this subset has been shown [PPS06] to be solvable in polynomial-time.

In this chapter, we introduce an automata-based algorithm for solving the synthesis problem for LSCs, specifically. We will see that the worst case time complexity of the algorithm is linear in the product of states, edges and inputs.

5.1 Related Work

There have been considerable efforts to synthesize executable systems from scenario-based requirements [KM94, KMST96, LMR98, KGSB99, WS00, WSK03, UKM04, LDD06]. In many of these papers, the requirements are given using a variant of classical message sequence charts while the synthesized system is state-based. Although there are many common aspects to our work and these papers, the main distinguishing feature is that we consider synthesis from LSCs, which we have already seen is more expressive than most of the classical MSC variants.

Synthesis from LSCs was first studied in [HK02], and is tackled there by defining consistency, showing than entire LSC requirements are consistent iff they are satisfiable by a state-based object system. A satisfying system is then synthesized. This line of work was continued in [HKP05] which includes an implementation of a sound but not complete algorithm for Statechart synthesis. A game theoretic approach to synthesis from LSCs involving a reduction to parity games is described in [BS03a, BHS05]. Synthesis from LSCs using a reduction to CSP is described in [SD05]. All the above papers were either theoretical and did not include an implementation, or the synthesis approach was sound but not complete, or the initial experimental results were not encouraging. An alternative strategy for synthesis from LSCs uses a translation from LSCs to temporal logic [KHP$^+$05] or automata [KW01] and then applies existing synthesis algorithms, e.g., [PR88, Var95, JB06].

5.2 Smart Play-Out

The Play-Engine [HM03] is a software package which enables users to create and execute LSC requirements based on the so-called play-in/play-out approach [HM03]. Play-in provides an intuitive method to capture requirements by manipulating the graphical interface of the system, while play-out executes the scenarios in a way that gives a feeling of running an implementation of the system.

An improvement called *smart play-out*, is introduced in [HKMP02]. Smart play-out uses verification methods—in particular, model-checking—to run LSC requirements and avoid certain LSC violations that may occur for the original version of play-out. One of the shortcomings of smart play-out is that the executable traces it generates are not guaranteed to avoid all violations of the LSC requirements.

The following results overcome the limitations of smart play-out by guaranteeing non-violating behavior for *all* scenario executions. We consider requirements in which certain objects participating in the scenario represent the system to be constructed and the rest represent an external environment. We assume *open systems* which assume no environment cooperation in satisfying the requirements.

Unlike [KHP$^+$05], we intentionally shift much of the LSC logic from a temporal logic formula to the finite discrete system so as to potentially reduce synthesis time complexity and focus our attention on a simpler generalized Büchi temporal logic formula that has a better chance of success in practical settings.

We view the synthesis problem over LSC requirements as a two-player game between the *system* and the *environment*. The system refers to the collection of software, hardware, or conceptual objects which are within the immediate control of the reactive system. Conversely, the environment refers to the collection of objects which are external to the program, and whose behaviors are beyond the control of the system.

The system attempts to *win the game* by taking whatever legal actions necessary to satisfy the requirements. Legal actions refer to a subset of the behaviors permitted by the requirements. The environment attempts to foil the system by preventing it from satisfying the requirements. Thus, the environment's goal is the negation of the system's. The environment has the upper hand because the system usually needs to meet numerous behavioral criteria in order to win, whereas the environment need only find one way to prevent the system from winning.

5.2.1 An Example

To illustrate why synthesizing LSC requirements is important, we present Fig. 5.1, consisting of two universal LSCs.

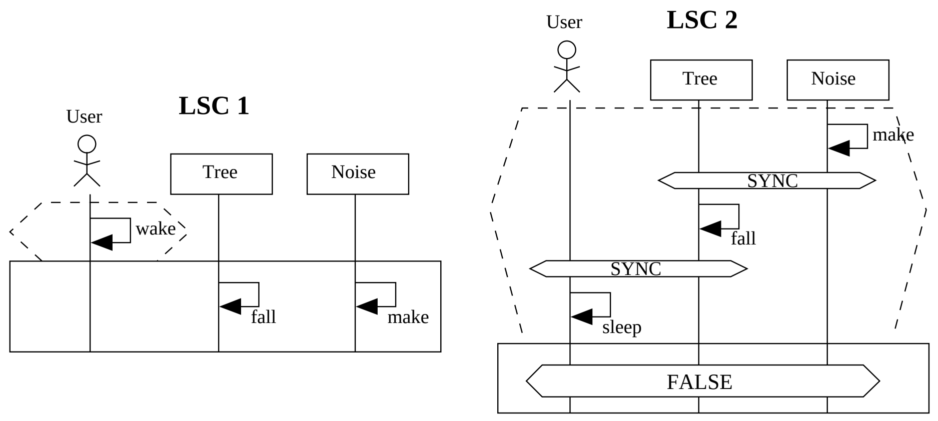


Figure 5.1: LSC Requirements for Smart Play-Out

Instance USER belongs to the environment, while TREE and NOISE are system instances. Accordingly, the behaviors of TREE and NOISE are within the control of the system we intend to construct, whereas the behaviors of the USER are assumed to be external.

According to LSC1, whenever USER sends the *wake* message, the system must respond with a non-deterministic ordering of message *fall* and *make* in order to satisfy the main chart. Therefore, the traces *wake*, *fall*, *make* or *wake*, *make*, *fall* are both acceptable for satisfying LSC1. On the other hand, LSC2 is an anti-scenario which states that the sequence *make*, *fall*, *sleep* cannot ever occur. The synchronizing conditions remove the otherwise non-deterministic ordering of *make*, *fall*, *sleep* to ensure that only traces with this precise ordering will satisfy the prechart.

We now consider how smart play-out might respond to an input message *wake*.

After formulating a model-checking problem that attempts to verify the non-existence of a satisfying LSC1 trace, the resulting counter-example yields one of the two possible event sequences above. Supposing that smart play-out executes the sequence: *make*, *fall*, LSC1 would be satisfied, but the prechart of LSC2 would advance such that the next enabled message is *sleep*. The environment could then execute *sleep* and violate the requirements. This illustrates the inability of smart play-out to look ahead into the future by more than one super-step.

Using synthesis, it is possible for the system to decisively choose an alternative sequence that would not allow the environment to violate the requirements. In response to the message *wake*, the synthesis algorithm would have removed the transitions that permit the sequence *make*, *fall* to occur, leaving *fall*, *make* as the only existing path.

5.3 Overview

The synthesis problem is broken down into two parts. The first part is the realizability question, which asks whether or not there exists some way for the environment to win the game. If the answer is in the negative, then the system is said to be *realizable*. The second part of the problem is to formulate a *winning strategy* for the system, provided the system is realizable. The winning strategy is like a road map which guides the system in producing behaviors that will always satisfy the requirements.

Let $\mathcal{L}$ be multiple untimed LSC requirements over an object system $\mathcal{O}$. We first translate $\mathcal{L}$ into an automaton using the discrete-time multiple LSC construction of section 4.2.2. Note that the untimed construction is just a specialized case of discrete-time, whereby the set V_T of ts-vars is empty. Recall that we solve the synthesis problem by formulating a two-player game between the system and environment. For the purposes of synthesis, the automaton must be converted into what we call a *game structure*, suitable for the purposes of game play. The two models are substantially the same, except that the transitions of the game structure are partitioned into two disjoint sets, necessary for identifying the transitions of each player.

5.4 Game Structures

A game structure is defined by $\mathcal{G} : \langle S, \Sigma, \Theta, \rho_e, \rho_s, \varphi \rangle$, where:

- S is a set of *states*.
- Σ is the set of *events*.
- Θ is the set of *initial states*.
- $\rho_e, \rho_s \subseteq S \times \Sigma \times S$ are the set of environment and system transitions.
- $\varphi \subseteq S$ is the *accepting set*.

Notice that there exist two transition sets, ρ_e and ρ_s, whose purpose is to identify the legal moves for each player at any given state. The intuition is similar to a board game such as Chess: just as Chess players are only permitted to move their own pieces, we similarly require that each player can only move between states using their own transitions.

The process of identifying which transitions belong to which player is carried out as follows. Given an object system $\mathcal{O}$, recall that each object $O \in \mathcal{O}$ is owned by either the system or environment. That is, $O.own \in \{sys, env\}$. We partition the objects of $\mathcal{O}$ into two disjoint sets according to their ownership. Using this information, we assign ownership to LSC messages. Specifically, messages *sent* by the environment are classified as environment messages, whereas those sent by the system are system messages. Since messages are defined, in part, by the LSC locations to which they are associated, we can map these locations to visible events by means of function $evnts$.

To summarize the above: by partitioning the set of objects, we arrive at a partitioning of the set of visible events. Let Σ_e be the set of environment events, and Σ_s be the set of system events induced by this object partitioning. If we let $\Sigma = \Sigma_e \cup \Sigma_s$, we can identify every transition (s, e, s') as belonging to the environment if $e \in \Sigma_e$, or the system if $e \in \Sigma_s$.

Ownership of hidden events, on the other hand, depends on the owner of the preceding visible event, as discussed in section 2.8.6. Therefore, hidden event ownership is dynamic, whereas visible event ownership is static. Fortunately, we need not concern ourselves with hidden events since the semantics are handled implicitly by the automaton construction.

5.5 Game Rules

A *play* of game structure $\mathcal{G}$ is a sequence of states $s_1, s_2, \ldots$, satisfying:

1. Initiality: $s_1 \in \Theta$

2. Consecution: $\forall i\ \exists e : (s_i, e, s_{i+1}) \in \rho_e \cup \rho_s$

3. Reactivity: $(s_i, e_i, s_{i+1}) \in \rho_e$, for infinitely many i.

The behaviors of the game follow the ω-regular expression $(\Sigma_e \cdot {\Sigma_s}^*)^\omega$, where the environment executes an event in Σ_e and the system responds with a *super-step* ${\Sigma_s}^*$ consisting of zero or more events. Since the system may choose empty super-steps, event traces could consist of events entirely in Σ_e. However, the system is not permitted to block environment input by prolonging its super-step to infinitely many events—hence, the third requirement above which states that environment events occur infinitely often[1].

We also require super-steps to be atomic in the sense that the environment is not permitted to interrupt the super-step by injecting a Σ_e event. This is justified because in real-life systems, environment inputs are often buffered—for example, in a FIFO queue—while the system processes events at the front of the queue. The synchrony hypothesis allows us to ignore issues of timing and queue boundedness which are beyond the scope of the current work.

We also require that environment messages only appear in precharts[2]. To see why this is necessary, we first note that the system is responsible for choosing super-steps that satisfy the requirements. Allowing environment messages to appear in the main chart would have the effect of creating future obligations for the environment. However, in a real-life setting, the environment could trivially violate the safety properties by producing the wrong behavior or not producing any behavior at all, thereby leading to

[1]One may model "no environment input" using a self-loop labeled by a reserved no-op symbol.

[2]This assumption is necessary for the current result, but not necessary in general. We later discuss a winning condition which does not require this assumption.

unrealizable requirements.

To remedy the above problem, one might impose artificial assumptions of the environment which state that the environment will cooperate by producing the expected behavior whenever a main chart requires the behavior to occur. However, since the purpose of this work is to underline the hostile and unpredictable nature of the environment, we felt it counterintuitive to take this approach.

5.6 The Synthesis Problem

Let $\mathcal{G}$ be a game structure with accepting set φ. We say that a game structure $\mathcal{G}$ *satisfies* φ, written $\mathcal{G} \models \varphi$ iff for all paths of $\mathcal{G}$, some state in φ occurs infinitely often. Formally, the synthesis problem is defined in two parts:

1. Realizability: Given a game structure $\mathcal{G}$ with accepting set φ, does there exist $\mathcal{G}'$ with $\mathcal{G}'.\rho_s \subseteq \mathcal{G}.\rho_s$ such that $\mathcal{G}' \models \varphi$?
2. If so, construct $\mathcal{G}'$.

In the sections to follow, we present a decision procedure for realizability checking. Later, we show how to arrive at the winning strategy, provided the specification is realizable. Construction of the strategy amounts to deciding which transitions, if any, to remove from $\mathcal{G}$.

5.6.1 Definitions

We say that a state $s \in S$ is *stable*, denoted $stable(s)$ iff $\forall i : [e_i \in \Sigma_e \Rightarrow \exists s' : (s, e_i, s') \in \rho_e]$. That is, iff every environment input is enabled from s.

Stable states are precisely those in which all LSCs are located somewhere within a prechart. This follows from our above assumption that environment messages cannot appear in the main chart—indeed, if any of the LSCs were in a main chart, then no environment inputs would be enabled. Hence, all LSCs are in their prechart.

A *super-step path by* e is a path between two stable states s_1, s_n (denoted $s_1 \overset{e}{\leadsto} s_n$), such that $s_1 \rightarrow \cdots \rightarrow s_n$ where $(s_1, e, s_2) \in \rho_e$ and for all $1 < i < n$, $(s_i, \_, s_{i+1}) \in \rho_s$. If we aren't interested in the identity of event e, we simply write $s_1 \leadsto s_n$.

A stable state s is *controllable* if for all $(s, e, s') \in \rho_e$ there exists a super-step path by e. State s is said to be *forward controllable to* $S' \subseteq S$ iff s is controllable and for all $s \leadsto s' : s' \in S' \wedge s'$ is controllable. Set $S' \subseteq S$ is a *forward controllable set* if each $s \in S'$ is stable and forward controllable to S'.

5.6.2 Main Result

Theorem 4 (Realizability) *Let $\mathcal{G}$ be a game structure where winning condition $\varphi = \{s \mid stable(s)\}$. Then there exists $\mathcal{G}'$ with $\mathcal{G}'.\rho_s \subseteq \mathcal{G}.\rho_s$ such that $\mathcal{G}' \models \varphi$ iff there exists a forward controllable set $S' \subseteq S$ in $\mathcal{G}$ s.t. $s_0 \in S'$.*

Proof:

$\Leftarrow$ Since there are finitely many states in S', all paths leaving s_0 must therefore lead to a cycle of stable states. Let S' be the set of states of $\mathcal{G}'$ and begin with the edges of $\mathcal{G}$. Let K be the set of states belonging to any super-step leading out of any state in S'. Remove from $\mathcal{G}'$ any edge which leads to a state outside of K. The remaining edges are guaranteed to lead to some other $s' \in S'$, since S' is a forward controllable set. Clearly, $\mathcal{G}' \subseteq \mathcal{G}$. Moreover, since all paths from s_0 lead to a cycle of stable states, we have $\mathcal{G}' \models \varphi$.

$\Rightarrow$ Let S' be the set of stable states in $\mathcal{G}'$ reachable from and including s_0. We show that this set is forward controllable: Since $\mathcal{G}' \models \varphi$, all paths leading out of states in S' lead to a cycle of stable states. Let C be the set of stable states appearing in any cycle. Since every state in C is in a cycle, each stable state in the cycle is controllable, since each leads to another stable state. Since every state in C is controllable, C must be forward controllable. Now consider the set of states $F \subseteq S'$ which aren't in C. Since $\mathcal{G}' \models \varphi$, these states all lead to C. Hence, $S' = F \cup C$ is forward controllable.

$\square$

5.6.3 Algorithm

We present a two-part algorithm for deciding realizability, written in pseudo-BASIC. The main idea is to determine the existence of a forward-controllable subset of S, which by Theorem 4, is equivalent to deciding realizability. The first part places tokens on states and propagates them across the game structure to determine all-pairs reachability among the stable states. In the second part, we identify uncontrollable states (stable states having no reachable successor), remove these states, determine whether this removal affects the controllability of other stable states, and repeat until we converge on the (possibly empty) set of forward controllable states.

In the algorithms below, ρ_e, ρ_s denote the transition relations for the environment and system, respectively. Each contain transitions of the form (s, e_i, s_i) where s is the origin state, s_i is the destination state and e_i is the event label, indexed by i. We let $stable(s)$ and $controllable(s)$ be assertions stating that state s is stable or controllable, repsectively.

Algorithm Part 1, presented in listing 5.1, contains two main loops. Within the first loop, *tokens* are placed on all states which are adjacent to a stable state. We assume a mapping of Σ_e to a finite subset of the naturals, such that every outgoing transition is assigned a number according to the outgoing transition's event. Each token $[s, i]$ contains information identifying both the identity of the stable state s and the number of the transition i from s which originated the token.

Listing 5.1: Algorithm Part 1: State Marking

```
Let $j$ := 0
Let $working(0)$ := $\{s \mid stable(s)\}$

For each $s \in working(j)$ and all $(s, e_i, s_i)$ in $\rho_e$
  Place token $[s, i]$ on state $s_i$
  If Not $stable(s_i)$ Then $working(j+1)$ := $working(j+1) \cup \{s_i\}$
  j++
End For

Until $working(j) = \emptyset$ Do:
  For each $s \in working(j)$ and all $(s, e_i, s_i) \in \rho_s$
     If token $[s, i]$ is on $s_i$ Then continue
     Place token $[s, i]$ on $s_i$
     If Not $stable(s_i)$ Then $working(j+1)$ := $working(j+1) \cup \{s_i\}$
  End For
  j++
End Until
```

The second loop propagates the token to all reachable states in a breadth-first manner. The propagation terminates wherever a stable state is encountered or a state containing the same token is encountered. The latter case is indicative of a cycle. The finite number of edges of $\mathcal{G}$ assures termination. When the algorithm terminates, each stable state contains the tokens of every other stable state that can reach it.

Turning to issues of time complexity, the first loop places one token on each of the outgoing edges of the stable states. We therefore have $|S| \cdot |\Sigma_e|$ tokens. The second loop propagates these tokens around the game board, such that each token traverses each edge at most once. Thus, the worst-case time complexity of the algorithm is $O(|S| \cdot |\Sigma_e| \cdot |E|)$.

The tokens placed by the algorithm above are each of the form $[s, i]$ where s is the stable state from which the token originated and i is the number of an outgoing transition from s which maps to some input event. Every stable state s' has a set of tokens, called a *token set*, representing all of the tokens which reached s'. More specifically, if $[s, i]$ is in the token set of stable state s' then s has a super-step by i leading to s'.

Algorithm Part 2, shown in listing 5.2, identifies a forward controllable set if it exists. To accomplish this, the set of *abandoned tokens* is first identified. A token is considered to be abandoned if it appears in none of the token sets of any of the stable states. If $[s, i]$ is an abandoned token, then s is uncontrollable since the super-step by i leading out of s doesn't lead to any stable state. Then, any tokens which belonged to the token set of the uncontrollable state are now considered abandoned. The process repeats again until no more abandoned tokens exist.

The set of states S' which remain after the process is finished is a forward controllable set, as will be proven later. If the initial state s_1 is among the states in S', then the

LSC requirements are realizable by Theorem 4.

For listing 5.2, let $AllTokens$ be the set of all tokens on the game structure. That is, $\{[s_1, 1], \ldots, [s_1, k], \ldots, [s_n, 1], \ldots, [s_n, k]\}$. We let $Tokens(s)$ be the token set of s and $Tokens = \{Tokens(s) \mid stable(s) \ \wedge \ s \in S\}$ be the set of non-abandoned tokens. The set of abandoned tokens is therefore given by $AllTokens \setminus Tokens$. The function predicate $controllable(s) := \{[s, 1], \ldots, [s, k]\} \subseteq Tokens$ decides the controllability of state s. The algorithm is as follows:

Listing 5.2: Algorithm Part 2: Find Forward Controllable States

```
Let $M(0)$ := $AllTokens \setminus Tokens$
Let j := 0

While $M(j) != \emptyset$ Do

    For each $[s, i] \in M(j)$
        If Not $controllable(s)$ Then
            $M(j+1)$ := $M_(j+1) \cup Tokens(s)$
            $Tokens(s) := \emptyset$
        Fi
    End For
    j++

End While
```

Theorem 5 (Correctness) *Algorithm Parts 1 and 2 correctly decide realizability.*

Proof:

Part 1 is a token propagation scheme. It consists of two loops, one which initiates the placement of tokens from every stable state, and the second which propagates the tokens until either a cycle or stable state is encountered. Clearly, one of the two will be encountered on every path and so the algorithm terminates.

Algorithm part 2 maintains a set of *abandoned* tokens—those which do not appear on any stable state. We begin with $M(0) = \mathit{AllTokens} \setminus \mathit{Tokens}$. Inductively, for each token $[s, i]$ contained in $M(j)$, we determine whether s is controllable. If so, nothing happens. Otherwise, all tokens from $\mathit{Tokens}(s)$ are removed and placed into $M(j+1)$. The algorithm terminates when $M(j+1)$ is empty—termination is guaranteed since the set of tokens is finite.

Upon termination of the algorithm, we claim that the set of controllable states is also a forward controllable set. Formally, $\mathit{controllable}(s) \Leftrightarrow s$ is forward controllable:

$\Leftarrow$ If s is forward controllable then it is also controllable, by definition. That is, $\mathit{controllable}(s)$ holds.

$\Rightarrow$ If $\mathit{controllable}(s)$ holds then we need only show that s' is controllable. It would then follow that s is forward controllable. Supposing s' is not controllable, $\mathit{controllable}(s')$ does not hold and so s' has at least one abandoned token. Since s' has an abandoned token, the set of tokens appearing on s'—including that belonging to s—would be removed by the algorithm. Therefore the super-steps of s must lead to a stable state which is controllable: a contradiction. Therefore, s is forward controllable.

Due to this and Theorem 4, we conclude that LSC requirements are realizable iff $\Theta \subseteq \{s \mid \mathit{controllable}(s)\}$.

□

Turning our attention back to time complexity, algorithm 2 is comprised of an outer and inner loop. The outer loop iterates so long as there exist abandoned tokens. In the worst case, one abandoned token will be added to $M(j)$ on each iteration and all tokens will eventually be abandoned, yielding at most $|S| \cdot |\Sigma_e|$ loop iterations. The inner loop performs constant-time processing on each token, making the worst-case time complexity $O(|S| \cdot |\Sigma_e|)$.

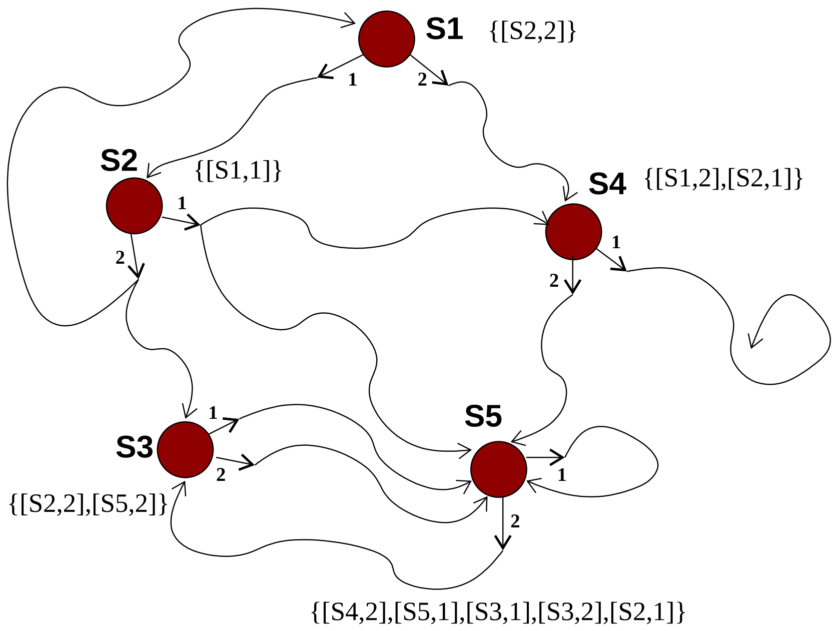
5.7 Example

To illustrate the above algorithm in action, consider Fig. 5.2, which depicts the state of affairs after the first part of the algorithm terminates. The darkened circles represent states. For the purposes of this example, we label the states $S1, \ldots, S5$. The straight arrows leading out of the stable states represent environment input transitions. We assume $|\Sigma_e| = 2$ and assign $1, 2$ to the outgoing transitions, which are shown next to each straight arrow. The curved arrows depict system responses, or paths consisting of unstable states (not shown).

Also depicted in Fig. 5.2 is the set of tokens belonging to each stable state. For example, state $S4$ has two tokens: $[S1, 2], [S2, 1]$, which means that state $S1$ has a super-step by transition 2 leading to $S4$ and $S2$ has a super-step by transition 1 also leading to $S4$. Notice that the super-step leading out of $S4$ by transition 1 leads to a cycle which never reaches any stable state. Consequently, token $[S4, 1]$ does not appear in the token set of any of the states and is therefore abandoned, as shown at the bottom.

Since $[S4, 1]$ is abandoned, it immediately follows that $S4$ is uncontrollable. The token set of $S4$ is therefore emptied. Each token which previously belonged to the token set of $S4$ is considered abandoned if that token appears on no other stable state.

The abandoned tokens correspond to the dashed super-steps shown in Fig. 5.3.



Abandoned: {[S4,1]}

Figure 5.2: Finding Uncontrollable States

Notice that there are no other super-steps leaving state $S1$ by transition 2, which means that $S1$ will be found to be uncontrollable in the next loop iteration. In contrast, the super-step by transition 1 leaving state $S2$ does have an alternate super-step, so the controllability of $S2$ is not affected.

When the algorithm terminates, it is left in the situation depicted in Fig. 5.4 in which all states remaining are forward-controllable. The LSC specification is realizable if the initial state is $S2$, $S3$ or $S5$.

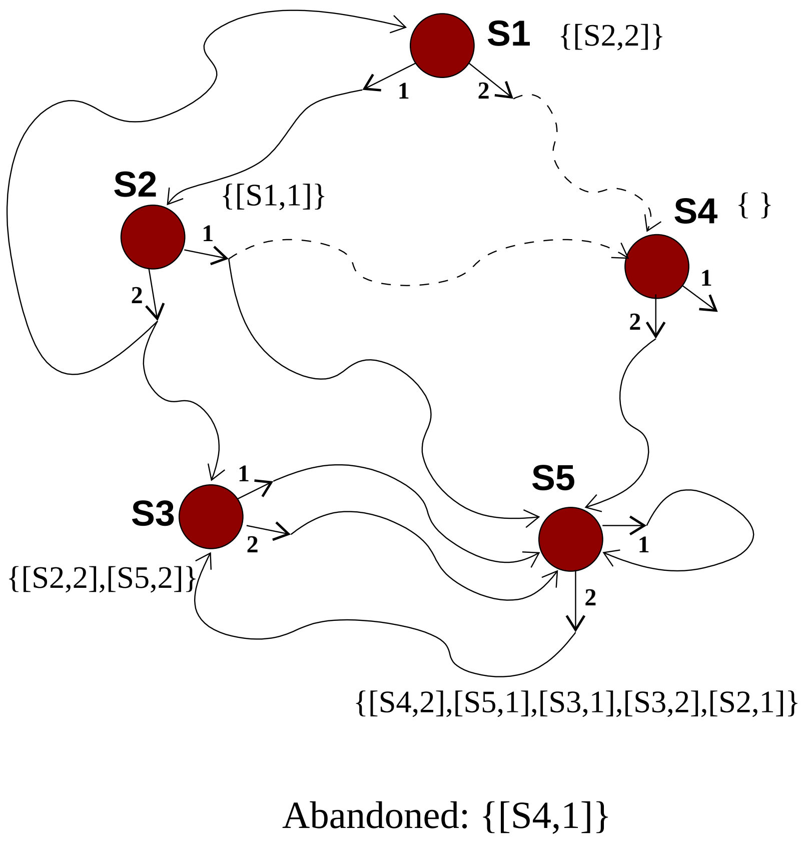


Figure 5.3: Removing Paths to Uncontrollable States

5.8 Finding the Winning Strategy

Once an LSC specification is determined to be realizable, the second part of the synthesis problem is to construct a game structure $\mathcal{G}'$ whose set of transitions is some subset of $\mathcal{G}$. Thus, the winning strategy amounts to determining which transitions to remove, if any.

The realizability decision procedure gave us a forward controllable set of states $S' \subseteq S$. Since every state $s \in S'$ is controllable, all paths out of s are guaranteed to lead to some other state in S'. Our goal is to cut out edges which *don't* lead to a state in S'.

To solve the problem, we place tokens on edges rather than states. More specif-

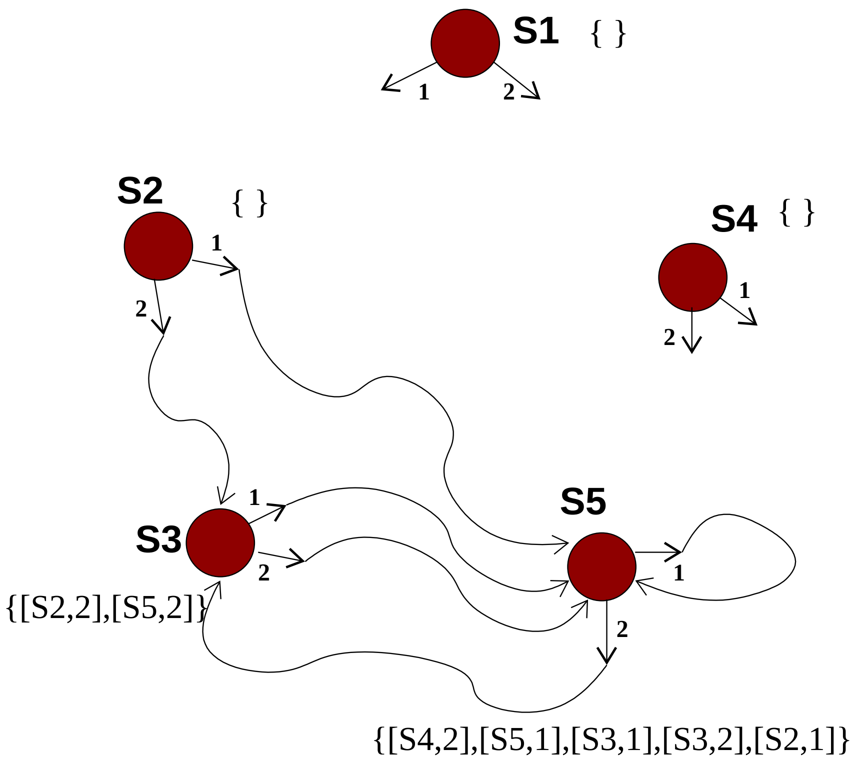


Figure 5.4: Forward Controllable Set

ically, from every state in $s \in S'$ we place a token on all edges which are *backward reachable* from s. Unlike the tokens seen previously, the edge tokens are not labeled in any way—an edge either has a token or it doesn't. Any path which intersects with a tokenized edge is guaranteed to lead to some $s \in S'$. It is also guaranteed *not* to lead to any state outside of S' due to the fact that the algorithm stops placing tokens once any stable state has been reached, whether the state is controllable or not. This would prevent an uncontrollable stable state from being encountered along the path to a controllable one. The algorithm is defined as follows:

Listing 5.3: Controller Extraction

```
Let j := 0
Let $backward(0) := \{s \mid controllable(s)\}$
```

```
Until $backward(j) = \emptyset$ Do

   For each $s \in backward(j)$ and all $(s_i, e_i, s) \in \rho_s$:
      Place a token on edge $(s_i, e_i, s)$
      If Not $stable(s_i)$ Then place $s_i$ in $backward(j+1)$
   End For
   j++

End Until
```

The time complexity of the algorithm is similar to the first algorithm presented, except that there are $|S|$ tokens instead of $|S| \cdot |\Sigma_e|$. The algorithm therefore has a worst-case time complexity of of $O(|S| \cdot |E|)$.

To arrive at the winning strategy, we remove from $\mathcal{G}$ all edges containing no token. By doing so, we arrive at game structure $\mathcal{G}'$, which implements the winning strategy.

Chapter 6

Symbolic LSC Synthesis

We presented an automata-based approach to LSC synthesis, which is a natural extension to the previously presented LSC-to-automaton construction. That approach lends itself to an *explicit state* construction, whereby every state and edge occupies space in memory. Another type of construction uses a *symbolic state* approach, whereby states and transitions are characterized by assertions over Boolean propositions. This can be represented efficiently in space using directed acyclic graphs called Binary Decision Diagrams (BDDs) [Bry86].

We now consider an alternate solution to synthesis of LSC requirements, in which the transition system for LSCs is encoded symbolically in the BDD-based language of SMV [McM92, McM98]. The synthesis algorithm itself is written in the SMV wrapper language TLV-BASIC, which is part of TLV [PS96]. The symbolic approach allows us to solve the same problem as before, but typically in a more space-efficient manner. This opens the door for implementing these methods in software with a much better chance of success.

We choose SMV and TLV, in particular, for compatibility with the smart play-out feature of the Play-Engine [HM03, HKMP02] which utilizes both languages. Smart play-out, being a closely related technology, provides much of the overhead necessary for our work. We can therefore refrain from duplicating many of the features already offered by the Play-Engine and focus more attention on the modifications and extensions necessary for synthesis. Aside from the synergies offered by smart play-out, we can also take advantage of several other Play-Engine features, most notably the graphical user interface and play-in.

As with the previous synthesis results, we view controller synthesis as a two-player open game consisting of an *environment* and *system*. We construct a *game structure* which encodes the rules of the game as a standard transition system, but this time using symbolic methods. Instead of using the realizability and winning strategy extraction methods from the previous chapter, we instead use the more general approach of [Pnu05] with some modifications. A practical implementation based on these results is also presented in [KPP07].

6.1 Symbolic Game Structures

The following definition is similar to that in [Pnu05]. A *game structure* (GS) is defined by $G : \langle V, X, Y, \Theta, \rho_e, \rho_s, \varphi\rangle$ consisting of:

- V, a finite set of typed *state variables*. We define a state s to be an interpretation of V, assigning to each variable $v \in V$ a value $s[v] \in D_v$ within its respective domain. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to expressions over V in the usual way. An *assertion* is a Boolean formula over V. A state s *satisfies* an assertion φ, denoted $s \models \varphi$, if $s[\varphi] = \text{TRUE}$. We say that s is a φ-state if $s \models \varphi$.

- $X \subseteq V$ is a set of *input variables* controlled by the environment. Let $\bar{X}$ denote the set of all input variable valuations.

- $Y = V \setminus X$ is a set of *output variables* controlled by the controller. Let $\bar{Y}$ denote the set of all output variable valuations.

- Θ is the initial condition characterizing all initial states of G.

- $\rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the environment. This is an assertion relating state $s \in \Sigma$ to a possible input value $\vec{x'} \in \bar{X}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_e identifies valuation $\vec{x'}$ as a possible *input* in state s, if for some output $\vec{y'}$, $(s, \vec{x'}, \vec{y'}) \models \rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ where $(s, \vec{x'}, \vec{y'})$ denotes a transition from state s to state $(\vec{x'}, \vec{y'})$.

- $\rho_s(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$ is the transition relation of the controller. This is an assertion relating state $s \in \Sigma$ to a possible output value $\vec{y'} \in \bar{Y}$ by referring to unprimed and primed copies of $\bar{X}$ and $\bar{Y}$. The transition relation ρ_s identifies valuation $\vec{y'}$ as a possible *output* in state s, if for some input $\vec{x'}$, $(s, \vec{x'}, \vec{y'}) \models \rho_e(\bar{X}, \bar{Y}, \bar{X}', \bar{Y}')$.

- φ is the winning condition, given by an LTL formula.

As can be seen, changes in state are characterized by changes in the variable valuations. Rather than partitioning transitions according to ownership as in the automata-based approach, we instead partition *variables* into those controlled by the environment (input variables) and those by the system (output variables). Each player may then observe and modify the valuations of its own variables, but can only observe the valuations of the opponent's.

6.1.1 Dependent vs. Independent Transitions

Following traditional game terminology, we say that a player *transitions* or *moves* whenever it modifies the variable valuations according to its transition relation. In most synthesis work, including [Pnu05], players strictly alternate between moving: a pre-designated player moves first according to the current valuation of the input and output variables. The second player then observes the same valuations as the first and also the first player's move and then moves herself. We refer to both moves as a *round*.

The work of [dAHM01] presents a different approach whereby both players move symmetrically and independently. That is, they move at the same time and both moves are a function of the current variable valuations only, but neither player may observe the opponent's move. We present a similar approach, whereby the players move symmetrically, as before. However, unlike the previous work, both players can move either dependently or independently according to our approach. A *dependent* move is one in which both players' moves are a function of the current variable valuations *and* the opponent's move. Otherwise, it is an *independent* move, like above.

We adopt this approach because LSCs inherently require objects to simultaneously synchronize at certain points during the execution. This happens, for example, when multiple instances move onto an LSC condition—either all the participating instances must move onto the condition, or none at all. Our definition lends itself to modeling this type of behavior quite naturally since all instances can observe whether the other instances are moving or not moving to a particular place during a given move. It is possible to simulate this type of synchronous behavior using the previous alternating-turn approach. However, this would require extra memory and logic.

To illustrate the notion of independent and dependent moves, consider the SMV code of listing 6.1.

Listing 6.1: Example SMV Code

```
1  next(sys) := case
2    sys=0 & env=0 : {1,2};
3    1 : sys;
4  esac;
5
6  next(env) := case
7    sys=0 & env=0 & next(sys)= 1 : 2;
8    sys=0 & env=0 : 3;
9    1 : env;
10 esac;
```

The example depicts the transition relation for environment input variable `env` between lines 1-4, and system output variable `sys` on lines 6-10. Elsewhere, variable `sys` is defined to range over $0, \ldots, 2$ and `env` over $0, \ldots, 3$.

The transition relation for each variable is expressed by a case statement. Each line of the case statement is of the form *expr : val* where *expr* is an expression over the variables and *val* is any number within the legal range. Each line is evaluated in the order appearing in the input. For each line, the variable, in the next state, will take on the value appearing on the right if the expression on the left holds. Otherwise, the next line is considered, and so on. Expression “1” is a catch-all expression referring to all cases not covered by the expressions appearing above it.

For example, according to line 2, if `sys` and `env` are both 0 in the current state, then `sys` can nondeterministically choose between 1 or 2 in the next state. Line 3 states that the value of `sys` remains unchanged for all other cases. Lines 2 and 3 are examples of independent moves, since neither relies on the environment's move (the value of `env` in the next state).

Moving to the transition relation for `env`, line 7 states that if `sys` and `env` are 0 and `sys` is 1 in the next state, then `env` is 2 in the next state. Line 7 is an example of a dependent move, since the case only holds with the cooperation of the system. Line 8 is also dependent since it implies `next(sys) != 1`. However, line 9 is independent because it doesn't depend on any particular value of `next(sys)`.

6.1.2 Deadlock

Conceptually, each round of play proceeds as follows: from a given state, both players each choose among any available move, including dependent moves. The players then simultaneously "announce" their intended move. There are two outcomes: if at most one of the players takes a dependent transition, the players may then proceed to move according to their announced choices. If both players choose dependent moves, there exists the possibility that the players' choices will conflict in such a way that neither player can proceed using their original choice. We refer to this phenomenon as a *deadlock*. In this situation, the players must repeat the selection process.

To illustrate the notion of deadlock in this context, consider the SMV code example below.

Listing 6.2: Deadlock Example

```
1  next(sys) := case
2    sys=0 & env=0 & next(env) = 1 : 1;
3    1 : sys;
4  esac;
5
6  next(env) := case
7    sys=0 & env=0 & next(sys)= 1 : 0;
8    1 : env;
9  esac;
```

First note that lines 2 and 7 both refer to dependent transitions, since the transition relation for each player's variable depends on the opponent's move. Now consider the state where `sys=0 & env=0` holds. According to line 2, if `env` is 1 in the next state, then `sys` is 1 in the next state. However, according to line 7, if `sys` is 1 in the next state, then `env` is 0 in the next state: a deadlock.

In the actual implementation, the players don't go through multiple rounds of selection in this way. Rather, each round is a member of the intersection of non-deadlocking moves. As we will see, our definitions follow very closely from those seen in [Pnu05, PPS06] with special deadlock checking added in.

6.2 Definitions

Let $\mathcal{G}$ be a game structure and s and s' be states of $\mathcal{G}$. We say s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We freely switch between $(s, \vec{x'}, \vec{y'}) \models \rho_e$ and $\rho_e(s, \vec{x'}, \vec{y'}) = 1$ and similarly for ρ_s.

A *play* σ of $\mathcal{G}$ is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* ($s_0 \models \Theta$) and *consecution* (for each $i \geq 0, s_{i+1}$ is a successor of s_j). Let σ be a play of $\mathcal{G}$. From state s, the environment chooses an input $\vec{x'} \in X$ and system chooses an output $\vec{y'}$ such that $\rho_e(s, \vec{x}', \vec{y}') = 1$ and $\rho_s(s, \vec{x}', \vec{y}') = 1$.

We say that play σ is *winning for the system* if it is infinite and satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is a function $f : \Sigma^+ \times \bar{X} \mapsto \bar{Y}$ such that if $\sigma = s_0, \ldots, s_n$ then for every $\vec{x}' \in \bar{X}$ and some $\vec{y}' \in \bar{Y}$ such that $\rho_e(s_n, \vec{x}', \vec{y}') = 1$ we have $\rho_s(s_n, \vec{x}', f(\sigma \cdot \vec{x}')) = 1$. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \ldots, s_i, s_{i+1}[\bar{X}]) = s_{i+1}[\bar{Y}]$, where $s_{i+1}[\bar{X}]$ and $s_{i+1}[\bar{Y}]$ are the restrictions of s_{i+1} to variable sets X and Y, respectively.

Strategy f is winning for the system from state $s \in \Sigma$ if all s-plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_c the set of states from which there is a winning strategy for the system. $\mathcal{G}$ is said to be winning for the system if all initial states of $\mathcal{G}$ are winning for the system. In this case, we say $\mathcal{G}$ is *realizable* and we *synthesize* a winning strategy which is a working implementation for the system. Otherwise $\mathcal{G}$ is *unrealizable*.

6.2.1 Controllable Predecessors

A transition leading into a state s is said to be *controllable* if the system can force the game into s in one step. That is, if the system can choose an independent transition for which all environment transitions lead into p—or—if all of the system's dependent transitions (for any possible environment transition) lead to p. The disjunction of these two requirements can be logically simplified to the expression $\forall \vec{x}' \exists \vec{y}' [\Phi_p \rightarrow \Phi_c]$. When conjoined with a deadlock check, it forms the controllable predecessor equation shown in equation (6.1) below.

The *controllable predecessor* of an assertion p, denoted $\bigcirc\!\!\!\!\diamond\, p$, is defined by the following, where $\|p\|$ denotes the set of states characterized by assertion p:

$$\Phi_p = (s, \vec{x}', \vec{y}') \models \rho_e$$

$$\Phi_c = (s, \vec{x}', \vec{y}') \models \rho_s \wedge (\vec{x}', \vec{y}') \in \|p\|$$

$$\|\bigcirc\!\!\!\!\diamond\, p\| = \{s \mid \exists \vec{x}' \exists \vec{y}' [\Phi_p \ \wedge \ \Phi_c] \ \wedge \ \forall \vec{x}' \exists \vec{y}' [\Phi_p \rightarrow \Phi_c]\} \qquad (6.1)$$

The right side of the main conjunction is the same as that in [Pnu05] and the left side amounts to assuring the absence of a deadlock.

6.2.2 Realizability & Winning Strategy

Once the notion of controllable predecessor is in place, the decision procedure for realizability and the extraction of the winning strategy proceeds according to [Pnu05]. The work focuses on winning conditions which are recurrence properties, i.e., LTL formulas of the form $\square \diamond q$ for an assertion q. We restrict our attention to formulas of this form for the purposes of our work also.

A state s satisfies $\diamondsuit\, p$ (for some assertion p) if the system can force the environment to reach a p-state within a single step from s. Based on this pre-image operator, a set of *winning states* is computed according to the following fix-point equation [Tar55], where μ and ν denote minimal and maximal (resp.) fix-points:

$$W_c = \nu Z \mu Y. \diamondsuit\, Y \vee q \wedge \diamondsuit\, Z \tag{6.2}$$

Given a game structure $\mathcal{G}$, we can check realizability of G by testing $\overline{W_c} \cap \Theta = \emptyset$. If $\mathcal{G}$ is winning for the system, a winning strategy is extracted by removing controllable transitions which lead to states outside of W_c.

6.3 Smart Play-Out

In this section, we describe how to construct a game structure from untimed multiple LSC requirements in SMV. Since our work is based on smart play-out [HKMP02], we first give an overview of the translation from LSC requirements to SMV.

SMV is a language for specifying program behaviors using a symbolic representation of the transition system. A *module* is a transition system consisting of variables and a transition relation. SMV permits the use of multiple modules, which can be composed either asynchronously or synchronously. When an asynchronous program executes, SMV interleaves the execution of the modules arbitrarily but fairly. A synchronous program is one in which all modules transition together simultaneously. In either case, modules may observe the variables of other modules. However, variable values may only be modified by the module that owns the variable.

Smart play-out constructs a transition system which is sufficient for the purposes of model checking. The general setup is the same as in the previous chapter. There exist a set of *stable states*, from which the environment may produce an input. It is then up to the system to formulate a response, called a *super-step*, leading to another stable state.

Smart play-out finds super-steps on a local level by formulating a model checking problem for a given environment input. Roughly speaking, smart play-out tries to verify the property “no super-step leading to a stable state exists” with the hope that the property is false. If it is indeed false, the model checker produces a counterexample as a witness to the existence of the super-step. Smart play-out then feeds the counterexample to the Play-Engine so that the user may witness the super-step being carried out on the program.

In order to set up the model checking problem, smart play-out first constructs an SMV transition system whose allowable behaviors are consistent with the LSC requirements. The transition system and an LTL formula serve as input to the model checker. For the purposes of our work, we aren’t interested in the LTL formula since we are performing synthesis rather than model checking. However, several parts of the transition system can be reused for the purposes of synthesis.

6.4 Main Result

There are a few main differences between smart play-out’s transition system and our game structure. On a high level, smart play-out defines one module for every object in the requirements and composes the modules asynchronously. In contrast, the synthesis algorithm of [Pnu05] requires precisely two (2) transition systems—one for the system and one for the environment. Therefore, most of the work amounts to reorganizing the smart play-out transition system and adding extra variables specific to the synthesis. Rather than provide the full details, we will focus mainly on the differences. The interested reader may consult [HKMP02] for the specifics of the smart play-out construction. Our definitions below will follow in the general spirit of this work.

6.4.1 Variables

Let $\mathcal{O}$ be an object system and let $\mathcal{L} = L_1, \ldots, L_n$ over $\mathcal{O}$ be a set of LSC requirements. We construct a game structure $\mathcal{G}$ with a set of *input variables* belonging to the environment and *output variables* belonging to the system. We now specify the set of variables V by defining the input variables X and output variables Y. The input variables are as follows:

1. act_{L_i} is 1 when the main chart of LSC i is active, and 0 otherwise.

2. $msg^s_{O_j \rightarrow O_k}$ denotes the sending of a message from object O_j to object O_k in which $O_j.own = env$ (O_j belongs to the environment). The value is set to 1 at the occurrence of the send and is changed to 0 at the next state.

3. $msg^r_{O_j \rightarrow O_k}$ denotes the receipt of a message by object O_k from object O_j in which $O_j.own = env$. As in the case of sending, the value is 1 at the instant the message is received and changes to 0 in the next state.

4. l_{L_i,O_j} is the location of object O_j in the main chart of LSC i where $O_j.own = env$. The location number ranges over $0, \ldots, l^{max}$ where l^{max} is the last location of O_j

in the main chart of LSC i. This variable is meaningful only when act_{L_i} is 1.

5. $l_{pch(L_i),O_j}$ is the location of object O_j in the prechart of LSC i where $O_j.own = env$. Its value ranges over $0, \ldots, l^{max}$ where l^{max} is the last location of O_j in the prechart of LSC i. This variable is meaningful only when act_{L_i} is 0.

6. $gbuchi$ is an auxiliary variable used to reduce a Generalized Büchi winning condition to a Büchi winning condition.

7. $envreq$ is a variable that determines which of the environment's objects has control in the next step.

The output variables belonging to the system are given by:

1. $msg^s_{O_j \to O_k}$ denoting the sending of a message from object O_j to object O_k in which $O_j.own = sys$ (O_k belongs to the system).

2. $msg^r_{O_j \to O_k}$ denoting the receipt of a message by object O_k from object O_j in which $O_j.own = env$.

3. l_{L_i,O_j} is the location of object O_j in the main chart of LSC i such that $O_j.own = sys$.

4. $l_{pch(L_i),O_j}$ is the location of object O_j in the prechart of LSC i such that $O_j.own = env$.

5. $currobj$ is a number ranging over $1, \ldots, |\mathcal{O}|$, referring to the object $O_{currobj}$ that currently has control of the execution.

The active flags, (act_{L_i}, for all i) and the auxiliary variable $gbuchi$ are not properties of the environment specifically, although they are environment variables. These are examples of *bookkeeping variables*, whose values are a function of the variables of both players. The choice of ownership could therefore be arbitrary. However, we assign ownership of these variables to the environment in order to be conservatively safe.

For example, if there exists a subtle error in the transition relation of any of these variables, the environment would find a way to utilize the error to its advantage in order to win the game and deem the requirements unrealizable. This is positive because we are forced to deal with the error in such a case. We could have alternatively chosen the system as the owner instead, in which case an error in the definitions could lead to false realizability—a more dangerous situation, particularly in the case of safety critical systems.

The purpose of the remaining variables, $envreq$ and $currobj$ are explained below.

6.4.2 Transitions

Smart play-out constructs a transition system comprised of an asynchronous composition of SMV modules. Each module defines the behaviors of one object in the LSC requirements, consisting of a set of variables and a transition relation. When generating traces, the TLV-BASIC model-checking routine arbitrarily selects modules for execution one at a time. The corresponding variables are then updated according to the transition relation of the selected module. Intuitively, each module's (i.e., object's) transition relation permits the object to carry out the next behavior on the object's instance line, with respect to the object's present LSC location.

On the other hand, synthesis requires a synchronous game structure in which all objects (and associated transition relations) belonging to the system are grouped into a single system module, and likewise for the environment. This raises the question of how to deal with the multiple definitions for each variable. We now describe the solution.

Let φ_i be any variable belonging to object O_i in the smart play-out construction. The transition relation, according to [HKMP02], for φ_i takes on the form:

$$\varphi_i' = \begin{cases} c_1 & \text{if } \Omega_1 \\ \vdots & \vdots \\ c_n & \text{if } \Omega_n \end{cases}$$

where c_j is a constant and Ω_j is a conditional expression over the variables of all objects in $\mathcal{O}$. In our synthesis construction, we have:

$$\varphi' = \begin{cases} \varphi_1' & \text{if } currobj = 1 \\ \vdots & \vdots \\ \varphi_k' & \text{if } currobj = k \end{cases}$$

where k is the number of objects. Therefore, we may simulate the asynchronous behavior of the smart play-out transition system by manipulating the variable $currobj$. This variable is responsible for determining which object, among the system and environment objects, move in the next step.

Variable $currobj$ must necessarily be owned by either the system or the environment. It would seem that permitting just one player to determine the current objects for both itself and its opponent could result in an unfair advantage. To level the playing field according to our result, the system and environment choose among their respective objects, but the decision of when each player gets their turn to decide goes to the system. To prevent the system from starving the environment of any opportunity to move, we will require the system to yield control to the environment infinitely often.

Formally, let $O_1, \ldots, O_j$ be the set of objects belonging to the environment and $O_{j+1}, \ldots, O_k$ be those of the system. We let:

$$envreq' = \{1, \ldots j\}$$

The environment uses $envreq$ to non-deterministically choose which of its objects will be the next to move once given a turn. With this in place, the system selects the current object in the following way:

$$currobj' = \{envreq, j + 1, \ldots, k\}$$

Note that this permits the system to execute arbitrarily long super-steps, since it can just keep selecting values between $j + 1, \ldots, k$. However, the winning condition discussed in the next section will require that $currobj' \leq j$ infinitely often, causing all super-steps to be finite.

6.4.3 Initial & Winning Conditions

The initial condition, Θ, of our game structure is the set of states in which $gbuchi = 0, act_{L_i} = 0$ for all i, all message variables are set to 0, and all location variables are set to 0. The initial value of $envreq$ is not specified in Θ, so the choice is therefore non-deterministic.

The winning condition φ is the generalized Büchi LTL formula:

$$\Box \Diamond \bigwedge_{i=1}^{n} act_{L_i} = 0 \land \Box \Diamond \, currobj \leq j$$

which is equivalent to:

$$\Box \Diamond \, gbuchi = 0$$

where:

$$gbuchi' = \begin{cases} 1 & \text{if } gbuchi = 0 \\ 2 & \text{if } gbuchi = 1 \land \bigwedge_{i=1}^{n} act_{L_i} = 0 \\ 0 & \text{if } gbuchi = 2 \ \land \ currobj \leq j \end{cases}$$

The above winning condition ensures that a stable state (characterized by all main charts being inactive) is visited infinitely often and that all super-steps are finite. It assumes that no environment messages appear in a main chart—for this, a more expressive winning condition beyond the scope of this paper is necessary.

The reason environment messages in the main chart are problematic is that the system would have to rely on the environment to perform the correct behavior at precisely the moment when an environment message is encountered in the main chart. While it is possible to adopt an assumption stating that the environment will always cooperate in these particular circumstances, we feel that this is unrealistically restrictive in light of our underlying assumption that the environment is hostile.

6.5 Implementation & Results

We implemented the above result as an extension to the Play Engine [HM03]. The implementation is broken down into two main pieces: the construction of the game structure and the TLV-BASIC [PS96] synthesis algorithm, both previously described.

The appendix contains SMV code for the example in section 5.2.1, simplified from its original form for readability. In particular, listing 1 shows the controllable predecessor function as discussed in section 6.2.1 and listings 5 and 6 show the environment and system transition relations (combined to form the game structure) for the specific input described in section 5.2.1. The remaining listings are not new to this work [Pnu05] but are included for the reader's convenience.

Once the Play Engine generates the game structure, it spawns a TLV process and the synthesis algorithm executes only once. If realizable, the resulting controller resides in memory for the lifetime of the play-out session. There exists an IPC component for interfacing the Play Engine and the externally spawned TLV session.

We have gathered performance data to give the reader a general sense of the runtime performance based on our experimentation to date. Our main focus was the running time of the synthesis algorithm, taking into account the size of the input game structure. In the worst case, the number of game structure states is exponential in the number of LSC locations, causing the resulting runtime performance to fall within an exponential upper bound. This is due to the non-determinism induced by the partial ordering which is inherent in most LSC requirements. We may overcome the exponential space blowup to some degree by utilizing the symbolic representation, however, exponential

LSC	*Linearizations*	*# Elements*
1	2	3
2	1	6
N3	6	5
PR3	1	4
D4	1	13

Table 6.1: Input LSC Requirements Data

Test #	*LSCs*	*Realizability (s)*	*Extraction (s)*
1	1,2	2	2
2	1,2,N3	19	20
3	1,2,N3,PR3	29	*
4	1,2,N3,PR3,D4	106	*
5	1,2,D4	47	55
6	1,2,PR3,D4	72	*

Table 6.2: Synthesis Runtime Performance

time complexity is seemingly unavoidable within this paradigm.

The tests we conducted consist of various combinations of five (5) LSCs comprised of only two elements: messages and conditions. In order to give a sense of the non-determinism present in each LSC and the approximate sizes, we show the number of linearizations (legal possible orderings of events) and the total number of elements for each LSC in table 6.5. LSC 1 and 2 are those shown in the example of section 5.2.1. The purpose of LSC `N3` is to introduce non-determinism into the requirements. LSC `PR3` partially refines the behaviors of `N3` by narrowing the number of linearizations of `N3` from 6 to 3. LSC `D4` is a deterministic LSC with substantially more elements than the rest. Conservative performance metrics were gathered using an Intel-based 900MHz

PC running Windows 2000 and are shown in table 6.5. All time units are expressed in seconds. Note that the introduction of `PR3` into the requirements causes the requirements to become unrealizable. For this reason, benchmarks for controller extraction are not provided since a controller does not exist in these cases.

Chapter 7

Contributions and Future Work

7.1 Summary

In this work, we have presented a method for synthesizing correct reactive programs directly from formal requirements. This work was motivated by the need to reduce implementation errors which typically arise during manually-driven development efforts. Toward achieving this goal, we selected the inter-object language of Live Sequence Charts to serve as a formal language with which to develop requirements.

In this work, we define a subset of LSCs which we believe is sufficient for showing the usefulness of a synthesis method. We discuss methods for translating LSC requirements to formal models—specifically, deterministic Büchi automata and timed automata. We then use these models as a basis for introducing an automata-theoretic approach to controller synthesis. We later describe an alternate solution using symbolic methods, which we implemented and incorporated into an LSC development tool called the Play-Engine.

Previously, Play-Engine users may have occasionally encountered violations of the requirements while observing behavioral reactions to user-inputs. It is difficult to conclude whether the violations constituted an error in the requirements or whether the Play-Engine simply made a bad choice. Our work provides users with a means of guaranteeing the correctness of the response to input requirements, insofar as non-violation is guaranteed provided the requirements are realizable. We view our software implementation as a successful proof-of-concept study, upon which we wish to expand in the future.

7.2 Contributions

Our LSC to automaton translation process described in chapter 4 focuses on a subset of the LSC language which considers messages, assignments, conditions, LSC variables, and object properties. Our motivation for producing the untimed automata construction was to use it as a stepping-stone toward a later dense-time model. The untimed construction was carried out independently of any other work, primarily out of need because the literature lacked a full detailed construction.

Other untimed implementations existed at the time, most of which focusing on small subsets of the language. Among the notable work includes the theoretical foundation in [HK00a] which serves as a basis for translations to state-based systems. Another theoretic approach to the LSCs to automata translation problem is presented in [BH02], but uses a different interpretation for dealing with multiple LSC requirements. Both of these works predated the LSC definitions presented in the Play-Engine [HM03], which we regard as the “official” LSC language interpretation for the purposes of synthesis.

Turning to the translation of timed LSCs, our work is based on [PGZ05]. The only other closely related work we know of which presents a constructive solution is presented in [KW01]. An interesting aspect of this construction focused on the method for which timing constraints were expressed, and the construction details pertaining to this. To this extent, the above work differs substantially from ours, since it uses a different means of expressing time constraints. Specifically, it borrows the idea of *timing annotations* from Symbolic Timing Diagrams, whereas we use the more freestyle approach of using assignments and conditions used by the Play-Engine. Additionally, our construction shows how to deal with multiple-LSC requirements, a critical aspect of the LSC language overlooked by most other work.

We also present the notion of warm temperature in LSCs. Here, we acknowledge two very reasonable semantic interpretations which previously existed for cold temperature and bring these distinctions out into the open. We propose to make use of the cold vs. warm temperature distinction by permitting users to utilize either interpretation in main charts as they see fit. We argue, by way of an example, that such a language addition could serve to make the LSC language more succinct by requiring possibly fewer LSCs for the same requirements. It should be noted that other independent work was carried out in [HM07], which considers a similar concept within the realm of UML 2.0.

Turning to the synthesis problem, this work presents two solutions to the same problem. The first contribution is an automata-based solution to controller synthesis, whose framework is well-suited for LSC requirements. This work provides an abstraction which focuses on major concepts such as *stable states* and *super-steps* without considering specific construction details, which may vary depending on the desired game rules. The details are encoded by means of states and transitions, depending on the LSC subset or specific semantic interpretation desired.

The second synthesis result was motivated by our goal to create a practical working implementation. In joint work with Hillel Kugler and Amir Pnueli, we created an extension to the Play-Engine's smart play-out feature. Whereas the synthesis result above focused more on the theory of controller synthesis for LSCs under Büchi winning conditions, this result utilizes a known implementation by Pnueli [Pnu05] and focuses more on the construction of the game structure—that is, precisely the detail not considered by the first solution. Additionally, we presented a modification to Pnueli's algorithm which follows a synchronous approach to environment/system interaction, in contrast to the existing alternating approach which is more commonly used.

7.3 Future Directions

Our work motivates the need for using formal requirements and demonstrates a method for automatically obtaining correct software from those requirements. However, in order to put these methods to use in practice, much more work is necessary. During the course of our research, we have identified several opportunity areas which we believe would best serve as a starting point for future work.

7.3.1 Time Consumption

This work and most literature on the subject make use of the synchrony hypothesis. Adoption of this hypothesis permits us to abstract away unnecessary details and work in an idealized world where nothing outside the realm of our system goes wrong. This is achieved by asserting, quite simply, that that all events occur instantaneously.

Even in a world where everything goes right, there is still no escaping the time-driven nature of software development. Function calls, for example, always consume time. It is natural to interpret LSCs messages as calls to object-oriented methods, where the sender represents the invoking object and the receiver represents the object whose interface is being invoked. As it currently stands, however, the act of receiving a message—which one may reasonably interpret to mean executing a method—occurs instantaneously. While assignments and conditions could be used for modeling the usual time consumption of functions, this method can be cumbersome. Object-oriented LSC modeling is discussed in greater detail in [HK00b].

In [EWY98], an extension to timed automata is proposed, whereby events are associated with real-time *tasks* which execute in parallel alongside the existing timed automaton in an event queue according to a predetermined scheduling policy. While the events themselves still execute in zero-time, the tasks which are spawned by these events require the passage of time before they can be removed from the queue. The question of whether it is possible for all tasks to terminate by predetermined deadlines is answered in the above work.

We believe that expanding our existing results to incorporate the so-called *timed automata with tasks* could be a rewarding line of work, particularly for the practitioner who wants to model real-time systems with time deadlines. This would involve performing a translation from LSC requirements to the timed automata with tasks model and a study of how synthesis could be performed using this model.

7.3.2 System Choices

Both synthesis results presented in this work involve the use of history information to meet a winning condition, as well as removing transitions which result in bad things happening at some point in the future. However, after the controller has been extracted, the system may still be left with a non-deterministic choice of transitions, all of which satisfy the winning condition. The question becomes: which transition, when faced with multiple choices, should the system take? Although all available choices will satisfy the requirements, some transitions may be more suitable than others, depending on the criteria one wishes to optimize.

Attributes of super-steps, such as minimal number of events, minimal battery consumption, or minimal time elapsed, could possibly be used. The issue isn't a lack of method for solving these problems. More to the point, it isn't immediately clear where some of the information would originate. For example, how does one estimate battery consumption on a transition by transition basis, and can it even be estimated faithfully?

7.3.3 Unrealizability

We previously discussed two methods for deciding the realizability of LSC requirements. In the case of unrealizable requirements, there are no methods currently in place for demonstrating the cause of unrealizability to the user. Currently, identification of the root cause requires patience along with an expert intuition about synthesis.

One step in the right direction might be to formulate a new realizable problem and reverse the roles of the environment and system: instead of asking whether the system can win the game, we instead ask if the environment can win. If the answer is in the affirmative, we propose using the Play-Engine to show traces which lead to winning behavior for the environment—or equivalently, losing behavior for the system.

This is only a first step, however. Counter-example traces can sometimes be long and incomprehensible, making it difficult to pinpoint the root cause of the problem. This could be especially problematic for synthesis, since the user would need to worry about *how* the program changes state from step to step as in verification, but also *who* causes the state change. We therefore consider it promising to pursue the avenue of conveying counter-example information to users in a more intuitive way.

7.3.4 Synthesis with Time

There has been previous work on the topic of controller synthesis from timed automata [AMPS98], and also synthesis of dense time systems using symbolic methods [AMP95]. A natural extension of our work would be to implement a similar result to what we already have, except using the dense-time case.

There exist a few issues which prevent us from plugging in our timed automaton construction as input to these algorithms. As previously mentioned, smart play-out and our untimed synthesis result utilize the SMV language, which is incapable of expressing dense time directly. Extra work would therefore be necessary to faithfully simulate a dense-time computation in a discrete time setting. This topic is discussed in greater detail in [GP00]. Secondly, we've already pointed out that time in general requires a more expressive winning condition than Büchi, since the environment would play a significant role in determining the passage of time. Given too much control, the environment could trivially violate timed LSC requirements by manipulating time passage to violate a main chart, for example. Just as we needed to impose reasonable environment restrictions on our untimed construction, we suspect that similar restrictions would be necessary for the dense time case.

Appendix

Listing 1: Controllable Predecessor

```
Func cpred(assert);

  Let assert1 := (( rho1 -> (rho2 & prime(assert)) ) forsome prv2 ) forall prv1;
  Let assert2 := (( rho & prime(assert) forsome prv1) forsome prv2);
  Return assert1 & assert2;

End -- Func cpred(assert);
```

Listing 2: Winning States

```
Func win(q);
   Local y;
   Local zz := 1;
   Local cy;
   Let iz := 1;
   Fix (zz)
     Let y  := 0;
     Let cy := 1;
     Fix (y)
       Let y := cpred(y) | q & cpred(zz);
       Let y[cy] := y;
       Let yy[iz][cy] := y;
       Let cy := cy + 1;
     End -- Fix (y)
    Let zz := y;
      Let z[iz] := zz;
     Let iz := iz + 1;
     Let maxcy := cy - 2;
   End -- Fix (zz)
   Return zz;
 End -- Func win(q);
```

Listing 3: Realizability Checking

```
 To check_realizability;
   prepare_synt;
   Let winning := win(recurrence);
   Let counter := Theta & !winning;
   If (counter)
     Print "Requirements are unrealizable.\n";
     Print counter;
     Return;
   End -- If (counter)
   Print "Requirements are realizable\n";
 End -- To check_realizability;
```

Listing 4: Controller Extraction

```
To synthesize;
  Let trans12 := rho1 & rho2;
  Let z := win(recurrence);
  Let trans := z & trans12 & recurrence & next(z);
  Let low := y[1];
  For (r in 2...maxcy)
    Let trans := trans | y[r] & !low & trans12 & next(low);
    Let low := low | y[r];
  End -- For (r in 2...maxcy)
  Let cts := new_ts(); -- Define new FDS
  Call set_V(vars, cts); -- Set variables
  Call set_I(Theta & z, cts);  -- Set initial condition
  Call add_disjunct_T(trans, cts); -- Set transition relation
End -- To synthesize;
```

Listing 5: Input (Environment) Module

```
MODULE Input (fall,make,loc_Sys1_LSC1,loc_Sys2_LSC1,
              loc_Sys1_LSC2,loc_Sys2_LSC2,current_process)

VAR

wake : boolean;    -- the "wake" message
sleep : boolean;  -- the "sleep" message
gbuchi : 0 .. 2;  -- auxiliary variable
active1 : boolean;  -- 1 when LSC 1 main chart is active
active2 : boolean;  -- 1 when LSC 2 main chart is active
loc_Env_LSC1 : 0 .. 2;
loc_Env_LSC2 : 0 .. 4;
envreq : 0 .. 0;  -- only 1 environment object (0)

ASSIGN

init(wake) := 0;
next(wake) := case
       next(current_process) != 0 : 0;
       next(fall)=0 & next(make)=0 & next(sleep)=0 & active1=0 : {0,1};
       1 : 0;
esac;

init(sleep) := 0;
next(sleep) := case
       next(current_process) != 0 : 0;
       next(wake)=0 & next(fall)=0 & next(make)=0 & active2=0 : {0,1};
       1 : 0;
esac;

init(gbuchi) := 0;
next(gbuchi) := case
       gbuchi=0 : 1;
       gbuchi=1 & active1=0 & active2=0 : 2;
       gbuchi=2 & current_process < 1 : 0;
       1 : gbuchi;
esac;

init(active1) := 0;
next(active1) := case
   active1=0 & next(loc_Env_LSC1)=1 : 1;
   active1=1 & next(loc_Env_LSC1)=1 & next(loc_Sys1_LSC1)=1 &
```

```
       next(loc_Sys2_LSC1)=1 : 0;
       1 : active1;
 esac;

init(active2) := 0;
next(active2) := case
     active2=0 & next(loc_Env_LSC2)=2 & next(loc_Sys1_LSC2)=2 &
        next(loc_Sys2_LSC2)=3 : 1;
--          There is no way to transition back to an inactive state
     1 : active2;
 esac;


init(loc_Env_LSC1) := 0;
next(loc_Env_LSC1) := case
     active1=0 & loc_Env_LSC1=1 : 0;
     next(current_process) = 0 & loc_Env_LSC1 = 0  & next(wake)=1 : 1;
     1 : loc_Env_LSC1;
 esac;

init(loc_Env_LSC2) := 0;
next(loc_Env_LSC2) := case
     active2=0 & loc_Env_LSC2 > 3 : 0;
     next(current_process)=0 & loc_Sys2_LSC2=2 & loc_Env_LSC2=0 : 1;
     next(current_process)=0 & loc_Env_LSC2=1 & next(sleep)=1 : 2;
     next(current_process)=2 & loc_Env_LSC2=0 & next(loc_Sys2_LSC2)=3 : 1;
     next(current_process)=0 & loc_Env_LSC2=2 & loc_Sys1_LSC2=2 &
         loc_Sys2_LSC2=3 : 4;
     next(current_process)=1 & loc_Env_LSC2=2 & next(loc_Sys1_LSC2)=4 : 4;
     next(current_process)=2 & loc_Env_LSC2=2 & next(loc_Sys2_LSC2)=5 : 4;
     1 : loc_Env_LSC2;
esac;

init(envreq) := 0;
next(envreq) := 0;
```

Listing 6: Output (System) Module

```
MODULE Output (wake, sleep, gbuchi, active1, active2, loc_Env_LSC1, loc_Env_LSC2, envreq)

VAR
fall : boolean;  -- the "fall" message
make : boolean;  -- the "make" message
loc_Sys1_LSC1 : 0 .. 1;
loc_Sys2_LSC1 : 0 .. 1;
loc_Sys1_LSC2 : 0 .. 4;
loc_Sys2_LSC2 : 0 .. 5;
current_process : 0 .. 2;

ASSIGN

init(fall) := 0;
next(fall) := case
    next(current_process) != 1 : 0;
    active1 = 1 & loc_Sys1_LSC1 = 0 & next(loc_Sys1_LSC1)= 1 &  next(wake)=0 &
         next(sleep)=0 & next(make)=0 : 1;
    1 : 0;
esac;

init(make) := 0;
```

```
next(make) := case
    next(current_process) != 2 : 0;
    active1 = 1 & loc_Sys2_LSC1 = 0 & next(loc_Sys2_LSC1)=1 & next(wake)= 0 &
          next(sleep)=0 & next(fall)=0 : 1;
    1 : 0;
esac;

init(loc_Sys1_LSC1) := 0;
next(loc_Sys1_LSC1) := case
    active1=0 & loc_Sys1_LSC1 >0 : 0;
    next(current_process) = 1 & loc_Sys1_LSC1 = 0  & next(fall)=1 : 1;
    1 : loc_Sys1_LSC1;
esac;

init(loc_Sys2_LSC1) := 0;
next(loc_Sys2_LSC1) := case
    active1=0 & loc_Sys2_LSC1 >0 : 0;
    next(current_process) = 2 & loc_Sys2_LSC1 = 0  & next(make)=1 : 1;
    1 : loc_Sys2_LSC1 ;
esac;

init(loc_Sys1_LSC2) := 0;
next(loc_Sys1_LSC2) := case
    active2=0 & loc_Sys1_LSC2 > 3 : 0;
    next(current_process)=1 & loc_Sys1_LSC2=0  & next(fall) = 1 : 1;
    next(current_process)=1 & loc_Sys1_LSC2=1 & loc_Sys2_LSC2 = 0 : 2;
    next(current_process)=2 & loc_Sys1_LSC2=1 & next(loc_Sys2_LSC2) = 1 : 2;
    next(current_process)=0 & loc_Sys1_LSC2=2 & next(loc_Env_LSC2)=4 : 4;
    next(current_process)=1 & loc_Sys1_LSC2=2 & loc_Env_LSC2=2 &
        loc_Sys2_LSC2=3 : 4;
    next(current_process)=2 & loc_Sys1_LSC2=2 & next(loc_Sys2_LSC2)=5 : 4;
    1 : loc_Sys1_LSC2 ;
esac;

init(loc_Sys2_LSC2) := 0;
next(loc_Sys2_LSC2) := case
    active2=0 & loc_Sys2_LSC2 > 4 : 0;
    next(current_process)=0 & loc_Sys2_LSC2=2 & next(loc_Env_LSC2)=1 : 3;
    next(current_process)=1 & loc_Sys2_LSC2=0 & next(loc_Sys1_LSC2)=2 : 1;
    next(current_process)=2 & loc_Sys1_LSC2=1 & loc_Sys2_LSC2=0 : 1;
    next(current_process)=2 & loc_Sys2_LSC2=1 & next(make)=1 : 2;
    next(current_process)=2 & loc_Sys2_LSC2=2 & loc_Env_LSC2=0 : 3;
    next(current_process)=0 & loc_Sys2_LSC2=3 & next(loc_Env_LSC2)=4 : 5;
    next(current_process)=1 & loc_Sys2_LSC2=3 & next(loc_Sys1_LSC2)=4 : 5;
    next(current_process)=2 & loc_Sys2_LSC2=3 & loc_Sys1_LSC2=2 &
        loc_Env_LSC2=2 : 5;
    1 : loc_Sys2_LSC2;
esac;

init(current_process) := { envreq, 1, 2 }; -- select: environment, Sys1, Sys2
next(current_process) := { envreq, 1, 2 };
```

Bibliography

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theor. Comp. Sci.*, 126:183–235, 1994.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 1989.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid System II*, volume 999 of *Lect. Notes in Comp. Sci.*, pages 1–20. Springer-Verlag, 1995.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[AT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Logic*, 5(1):1–25, 2004.

[BDK$^+$02] J. Bohn, W. Damm, J. Klose, A. Moik, and H. Wittke et al. Modeling and validating train system applications using statemate and live sequence charts. In H. Ehrig, B. J. Kramer, and A. Ertas, editors, *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*, 2002.

[BG92] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, 1992.

[BG02] A. Bunker and G. Gopalakrishnan. Verifying a VCI Bus Interface Model Using an LSC-based Specification. In H. Ehrig, B. J. Kramer, and A. Ertas, editors, *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, pages 1–12, 2002.

[BGP96] B. Bérard, P. Gastin, and A. Petit. On the power of non observable actions in timed automata. In C. Puech and R. Reischuk, editors, *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS'96)*, number 1046 in Lecture Notes in Computer Science, pages 257–268. Springer-Verlag, 1996. Full version in Technical Report 95.08 of ENS de Cachan, LIFAC.

[BH02] Y. Bontemps and P. Heymans. Turning high-level Live Sequence Charts into automata. In *"Scenarios and State-Machines: Models, Algorithms and Tools" SCESM workshop of the 24th Int. Conf. on Software Engineering ICSE*, 2002.

[BHK03] Y. Bontemps, P. Heymans, and H. Kugler. Applying LSCs to an Air Traffic Control Case Study. In *Proc. 2nd Int. Workshop on Scenarios and State Machines (SCESM'03)*, 2003.

[BHS05] Y. Bontemps, P. Heymans, and P. Y. Schobbens. From live sequence charts to state machines and back: A guided tour. *IEEE Trans. Software Eng.*, 31(12):999–1014, 2005.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[BMR83] G. Berry, S. Moisan, and J. Rigault. ESTEREL: Toward a synchronous and semantically sound high level language for real time applications. In *Proc. IEEE Real-Time Systems Symposium*, pages 30–37, Los Alamitos, CA, 1983. IEEE CS Press.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1999.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[BS03a] Y. Bontemps and P.Y. Schobbens. Synthesizing open reactive systems from scenario-based specifications. In *Proc. of the 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03)*. IEEE Computer Science Press, 2003.

[BS03b] A. Bunker and K. Slind. Property Generation for Live Sequence Charts. Technical report, University of Utah, 2003.

[Büc62] J.R. Büchi. On a decision method in resricted second-order arithmetics. In *Proc. Int'r Congr. Logic, Method. Phil. of Sci., 1960*, pages 1–12. Stanford University Press, 1962.

[dAHM01] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *Proc. 16th IEEE Symp. Logic in Comp. Sci.*, pages 279–290. IEEE Computer Society Press, 2001.

[DGP97] V. Diekert, P. Gastin, and A. Petit. Removing ϵ-transitions in timed automata. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97)*. Springer-Verlag, 1997.

[DH01] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1989.

[EWY98] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Nordic Workshop on Programming Theory*, 1998.

[GBGM91] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with signal. In *Proceedings of the IEEE, Special Issue*, September 1991.

[Gil03] A. Gilboa. Finding All Super-Steps in LSC Specifications. Master's thesis, Weizmann Institute of Science, Israel, 2003.

[GP00] O. Grinchtein and A. Pnueli. Dense-time analysis with fractional adjustment steps. Technical report, The John von Neumann Minerva Center, the Weizmann Institute, 2000.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har00] D. Harel. From play-in scenarios to code: An achievable dream. In Tom Maibaum, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lect. Notes in Comp. Sci.*, pages 22–34. Springer-Verlag, 2000.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proc. 18th Int. Conf. on Software Engineering*, Berlin, March 1996.

[HK00a] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. In *Proc. 5th Inf. Conf. on Implementation and Application of Automata (CIAA'00)*, volume 2088 of *Lect. Notes in Comp. Sci.*, pages 1–33. Springer-Verlag, July 2000.

[HK00b] D. Harel and O. Kupferman. On the behavioral inheritance of of state-based objects. In *Proc. 34th Int. Conf. on Components and Object Technology*, Santa Barbara, August 2000. IEEE Computer Society.

[HK02] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science (IJFCS).*, 13(1):5–51, Febuary 2002. (Also,*Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, July 2000, Lecture Notes in Computer Science, Springer-Verlag, 2000.).

[HK04] D. Harel and H. Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *Lect. Notes in Comp. Sci.*, pages 325–354. Springer-Verlag, 2004.

[HKMP02] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc.* 4^{th} *Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *Lect. Notes in Comp. Sci.*, pages 378–398, 2002. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.

[HKP05] D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenarios-Based Requirements. In *Formal Methods in Software and System Modeling*, volume 3393 of *Lect. Notes in Comp. Sci.*, pages 309–324. Springer-Verlag, 2005.

[HLN$^+$90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Engin.*, 16:403–414, 1990.

[HM01] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Tech. Report MCS01-15, The Weizmann Institute of Science, 2001. Submitted.

[HM02] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, Fort Worth, Texas, 2002.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[HM07] D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and System Modeling (SoSyM)*, 2007. To Appear. Early version in 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06).

[HMKT00] J.G. Henriksen, M. Mukund, K. Narayan Kumar, and P.S. Thiagarajan. On Message Sequence Graphs and finitely generated regular MSC languages. In *Proceedings of the 27th International Colloquium on Automata Languages and Programming (ICALP'2000)*, number 1853 in Lecture Notes in Computer Science, Geneva, Switzerland, 2000. Springer.

[HMNT99] J. Henriksen, M. Mukund, K. Narayan, and P. Thiagarajan. Towards a theory of regular msc languages, 1999.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI Series*, pages 477–498, New York, 1985. Springer-Verlag.

[JB06] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conferences on Formal Methods in Computer Aided Design (FMCAD '06)*, San Jose, California, USA, Nov 2006.

[Jes04] Anick Jesdanun. GE Energy acknowledges blackout bug. Associated Press, as published at http://www.securityfocus.com, February 2004.

[KGSB99] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Proc. Int. Workshop on Distributed and Parallel Embedded Systems (DIPES'98)*, pages 61–71. Kluwer Academic Publishers, 1999.

[KHK$^+$03] N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J.A. Hubbard, and M.J. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. In Corrado Priami, editor, *Proc. Int. Workshop on Computational Methods in Systems Biology (CMSB 2003)*, volume 2602 of *Lect. Notes in Comp. Sci.*, pages 4–20. Springer-Verlag, 2003. Extended version appeared in Modeling in Molecular Biology, G.Ciobanu (Ed.), Natural Computing Series, Springer, 2004 .

[KHP$^+$05] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In N. Halbwachs and L.D. Zuck, editor, *Proc. 11th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05),*, volume 3440 of *Lect. Notes in Comp. Sci.*, pages 445–460. Springer-Verlag, 2005.

[KM94] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software — Practice and Experience*, 24(7):643–658, 1994.

[KMST96] K. Koskimies, T. Mannisto, T. Systa, and J. Tuomi. SCED: A Tool for Dynamic Modeling of Object Systems. Tech. Report A-1996-4, University of Tampere, July 1996.

[KPP07] Hillel Kugler, Cory Plock, and Amir Pnueli. Synthesizing reactive systems from LSC requirements using the Play-Engine. In *OOPSLA Companion*, pages 801–802, 2007.

[KW01] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *Proc. 7th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci.*, Springer-Verlag,* 2001.

[LDD06] H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM'06)*, pages 5–12, 2006.

[LL93] Peter B. Ladkin and Stefan Leue. What do message sequence charts mean? In *FORTE*, pages 301–316, 1993.

[LMR98] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM models from message sequence chart specifications. Tech. Report 98-06, University of Waterloo, April 1998.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[Mar90] F. Maraninchi. Argos: a graphical synchronous language for the description of reactive systems. Technical report, C29, LGI-IMAG Grenoble, 1990. Submitted to *Science of Computer Programming*.

[McM92] K. L. McMillan. *The SMV System Draft*. Carnegie-Mellon Univerisity, 1992.

[McM98] K.L McMillan. Getting started with smv. Technical report, Cadence Berkeley Labs, 1998.

[MSC99] ITU-TS Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva, 1999.

[NAS99] NASA. Mars Climate Orbiter Mishap Investigation Board Phase I Report. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, November 1999.

[PGZ05] C. Plock, B. Goldberg, and L. Zuck. From requirements to specifications. In *Proc.* 12^{th} *Annual IEEE Intl. Conference and Workshop on the Engineering of Computer-Based Systems*. IEEE Computer Society Press, 2005.

[Pnu05] A. Pnueli. Extracting controllers for timed automata. Technical report, New York University, 2005.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006.

[PR88] A. Pnueli and R. Rosner. A framework for the synthesis of reactive modules. In F.H. Vogt, editor, *Proc. Intl. Conf. on Concurrency: Concurrency 88*, volume 335 of *Lect. Notes in Comp. Sci.*, pages 4–17. Springer-Verlag, 1988.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 652–671. Springer-Verlag, 1989.

[PS96] A. Pnueli and E. Shahar. The tlv system and its applications. Technical report, The Weizmann Institute, 1996.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.

[Ren95] M.A. Reniers. Syntax requirements of Message Sequence Charts. In *In R. Braek and A. Sarma, editors, Proc. of hte 7th SDL Forum*, pages 63–74. Elsevier Science Publishers B.V., October 1995.

[SD05] J. Sun and J. S. Dong. Synthesis of distributed processes from scenario-based specifications. In *International Symposium of Formal Methods Europe (FM'05)*, pages 415–431, 2005.

[Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. of Mathematics*, 5, 1955.

[UKM04] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Software Engin. Methods*, 13(1):37–85, 2004.

[Var95] M.Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *P. Wolper, editor,* Proc. 7^{th} Intl. Conference on Computer Aided Verification (CAV'95), *volume 939 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 267–278, 1995.

[WS00] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323. ACM Press, 2000.

[WSK03] J. Whittle, J. Saboo, and R. Kwan. From scenarios to code: an air traffic control case study. In *25th International Conference on Software Engineering (ICSE 2003)*, pages 490–495. IEEE Computer Society, 2003.

[Z1293] Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1993.

[Z1296] Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.